1994

Transaction management in object-oriented data base systems

Pinaki D. Shah
Iowa State University

Order Number 9424257

Transaction management in object-oriented data base systems

Shah, Pinaki D., Ph.D.

Iowa State University, 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Transaction management in object-oriented data base systems

by

Pinaki D. Shah

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Computer Science
Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1994

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I wish to thank my advisor Dr. Johnny Wong for his constant help and guidance during the course of this research. This research would not have been possible without his continuous encouragement and advice. I wish to thank him for introducing me to the areas of computer security, distributed systems and transaction management. In particular, I thank him for the numerous discussions on object-oriented data bases.

I would also like to thank Dr. Pete Boysen and Dr. Rex Thomas. I have gained valuable experience working with them on ABC. I would also like to thank Heekuck Oh for being a good friend and co-worker and for introducing me to the world of fishing. I wish to thank all the members of Dr. Wong's research group for the numerous discussions and advice during the course of my research. I wish to thank Rajesh Parekh for his help in the preparation of this dissertation. I would especially like to thank the Montgomery family (Pat, Barb, Kelly and Kaye) for being my family away from home.

Last but not the least, I would like to thank my parents and my sister for always supporting me. They have continually provided me with moral support, without which I would not have completed this research.

1. INTRODUCTION

1.1 Introduction

The newer generation of data base applications requires modeling techniques more powerful than the ones offered by relational systems. The *flat* nature of relational systems [1] imposes severe restrictions on application modeling. Object-oriented data bases provide a promising alternative for advanced applications such as computer aided design (CAD) and multimedia data bases (MMDB). However, many issues in object-oriented systems are still not well understood.

Transaction management in object-oriented data bases is one of the issues which requires further research. Transactions in applications such as CAD and MMDB systems differ from those in conventional applications in many respects. Some of these differences include the duration of transactions, granularity of transaction management features and complications arising out of various object-oriented characteristics such as class hierarchies, class-composition hierarchies, encapsulation and inheritance.

Query processing is another important issue in object-oriented data base systems. The inherently navigational nature of object-oriented data bases poses some problems for application users who prefer declarative query languages. Currently, some of the research in object-oriented data bases has been focussed on the definition of such

query languages [2, 3, 4, 5, 6]. However, most of this research does not provide an insight into how such user defined queries can be efficiently translated to appropriate data base operations.

The research work and published literature in the field of transaction management for object-oriented data base systems is still in its infancy. A suitable transaction model is considered to be one of the most important issues that remain unresolved. Past research in this area has taken the approach of simply extending the concepts developed in the relational framework to the object-oriented model. Unfortunately, such an extension tends to ignore the impact of the semantic richness of the object-oriented data model. Furthermore, the nature of transactions in applications such as CAD and MMDB systems is much different from that in conventional data base systems. These applications often involve long-duration cooperating transactions as opposed to the short-duration stand-alone ones in conventional data base systems. The complex interactions between transactions in such applications also requires alternate criterion for defining *correct* executions. The traditional notion of *serializability* is no longer appropriate in the presence of cooperating transactions.

Our research goals are to provide a model for transaction management in object-oriented data base systems and provide for underlying support to facilitate query processing. The transaction model employed in relational systems is not suitable for the advanced applications arising in object-oriented data base systems. In this research work, we have proposed a model that uses correctness criterion other than serializability; the details of our research goals will be discussed in the next section.

1.2 Research Goals

In the preceding section, we have described how transaction management in object-oriented data base systems differs from that in relational systems. These differences indicate the need for providing a better transaction model which takes into consideration the features of the object-oriented data base applications and their unique requirement. Our research goals are as follows:

- Provide an alternate transaction model for object-oriented data base systems.
- Devise concurrency control mechanisms for such a transaction model.
- Design mechanisms to allow efficient support for query processing in an object-oriented data base.

1.2.1 Alternate transaction model

Conventional data base systems use *serializability* as the correctness criterion for concurrently executing transactions. However, serializability is an inappropriate correctness criterion for advanced applications such as CAD and MMDB systems with long-duration cooperating transactions. Furthermore, the various features of the object-oriented data model such as the class and class-composition hierarchies, inheritance etc. give rise to additional complications. The complex nature of the object-oriented data model introduces interactions between data elements (objects) that are not found in the *flat* relational systems. For example, the notion of *conflict* in an object-oriented data base system needs to be redefined. In conventional data bases, two operations are said to conflict if they operate on the *same* data element and at least one of them is a *write*. However, in an object-oriented data base, two

operations may conflict even if they are not operating on the same data element (object) because an operation on an object may in turn invoke an operation on some other object.

When long-duration transactions cooperate and observe each others' intermediate results, serializability is not an appropriate correctness criterion. No serial execution of cooperating transactions may be equivalent to a concurrent execution of the transactions. Therefore, the transaction model for such advanced applications needs to address the issues that are typical to such applications. Furthermore, object-oriented data bases contain a high degree of semantic information. This semantic information can be used to facilitate transaction management. Our goal is to provide a new transaction model that exploits the semantic richness of and is appropriate for advanced applications in object-oriented data base systems.

1.2.2 Concurrency control mechanisms

Our transaction model for object-oriented data bases will provide a framework for advanced applications to interact with the data base. The correctness criterion for concurrent executions of transactions will be defined by the transaction model. The concurrency control mechanisms will then need to guarantee that concurrently executing transactions do not violate the correctness criterion.

Majority of the concurrency control mechanisms in conventional data base systems employ some form of *locks* or *time-stamps*. In the object-oriented context, our aim is to devise suitable mechanisms. The semantic richness of the object-oriented data model introduces additional complications. If locks are used, it is desirable that the locks be set at the object level. However, several issues need to be addressed. For

example, if an object O is locked, does it imply that all the objects which represent its instance variables are also locked? If there are multiple paths to access a shared object, how do we ensure that the locks are visible on all the paths.

As part of our research, we will devise concurrency control mechanisms which will ensure a correct execution of transactions. Wherever possible, it is our goal to exploit the semantic information about objects in the data base.

1.2.3 Support for query processing

Query Processing in object-oriented data bases is being investigated by several research teams [2, 3, 4, 6]. The primary emphasis of most of the current research is the specification of declarative queries. Users of relational data bases are more familiar with these kinds of queries (e.g., Structured Query Language - SQL). Therefore developers of object-oriented data bases are attempting to provide a similar query language for their environments.

The declarative user queries have to be translated to data base operations. In an object-oriented context, this would mean translating the queries to the underlying object-oriented language. Our goal is to identify mechanisms which will facilitate a translation from the declarative queries to operations on the object-oriented data base. We will identify a set of core features which are common to the various proposed query languages. Based on these features, we propose the kind of data base support which would ease their implementation. The user queries interact with the data base in the form of transactions. Thus the work in the area of query processing will tie in with our research on the transaction model and the concurrency control mechanisms.

1.3 Organization of the Dissertation

Chapter 2 introduces some of the object-oriented features which are the basis of most object-oriented systems. The basics of transaction management are also discussed. Chapter 3 introduces our transaction model for object-oriented data base systems. This model exploits the semantic richness of object-oriented data bases and is suitable for the long-duration cooperating transactions that characterize advanced applications such as CAD and MMDB systems. A concurrency control protocol for the transaction model is presented in Chapter 4. The protocol ensures that only *correctable* executions are allowed. Chapter 5 discusses the issue of query processing in object-oriented data bases. The underlying data base support necessary for efficient handling of user queries is discussed. Chapter 6 provides a few concluding remarks.

2. BACKGROUND

2.1 Introduction

As stated in the previous chapter, our research goals include providing a new transaction model for object-oriented data bases. In this chapter, we provide an overview of object-oriented systems, transaction management, concurrency control and other related issues. We also discuss some of the current literature in these areas and how it fails to meet the requirements of transactions occurring in advanced application such as CAD and MMDB systems.

Section 2.2 provides an introduction to the object-oriented paradigm and discusses various features that characterize object-oriented systems. The core concepts of objects, classes, and inheritance are described. An overview of transaction management and concurrency control is provided in Section 2.3. Section 2.4 describes the unique nature of transactions in advanced applications that use object-oriented data bases.

2.2 Overview of Object-oriented Features

Although researchers seem to agree that something beyond the relational model is needed for advanced data base applications, there is no consensus regarding what the object-oriented model should be. There exists no standard and universally ac-

cepted definition of what *object-oriented* really means. However, there seems to be agreement regarding some of the concepts which are considered to be *core* to object-oriented systems. Some of these basic concepts are described by Bancilhon [7] and Kim [5], among others.

2.2.1 Core object-oriented concepts

Of the many aspects which are characterized as being object-oriented, the following concepts have received general agreement by the research community:

- **Objects and Object Identifiers**

  In an object-oriented system, all real-world entities are represented as objects. Each object has a unique identifier. An object may be either a simple object or it could represent some complex entity. An object may have references to other related objects. Thus, an object can represent a large amount of semantic information.

  The specification of objects in terms of object identifiers contrasts with the value based specification of tuples/records in a relational system. This means that object-oriented systems are intrinsically navigational in nature. Therefore they cannot directly support declarative queries. Relational systems support declarative queries easily because of the value-based nature of the retrieval. However, object-oriented systems can use some indexing techniques to reduce the navigational nature of object manipulation.

- **Attributes and Methods**

  Objects can have one or more attributes, and methods which operate on the

values of these attributes. Unlike the relational system, where the values of the attributes are primitive data elements, the attributes in an object may themselves represent other complex objects. In addition, the attribute may represent a set of values, instead of just a single value. This again contrasts with the relational systems which do not allow for sets of values in a tuple.

Since an attribute of an object can be any arbitrarily complex object, it can lead to nested structures (as opposed to the flat structure of a tuple). The possibility of nesting is an important characteristic of the object-oriented model and is required by many applications including CAD.

- **Encapsulation and Message Passing**

  The object encapsulates both the attributes and the operations on these attributes. To access the values of these attributes, messages have to be sent to the object. External entities cannot directly access the attributes of the object. Thus, an object has a well-defined interface through which its attributes may be accessed.

- **Class**

  Classes provide a means to group objects which share the same set of attributes and methods. An object belongs to a class and is said to be an instance of that class. One could envision an object to be an instance of several classes, but for performance reasons, it may be desirable to restrict it to a single class. In order to treat everything uniformly as an object, the class itself is considered to be an instance of another class - the metaclass.

Queries in an object-oriented system are generally directed towards a class. Thus, the class provides an important means for grouping together objects, which can later be queried. Classes can also be used to enhance the semantic integrity of object-oriented data bases. As mentioned earlier, an object may have a number of attributes. These attributes are themselves objects and hence they belong to some class, giving rise to the class-composition hierarchy.

- **Class Hierarchy, Class-composition Hierarchy, and Inheritance**

  The concept of inheritance is a very important component of an object-oriented system. The classes in an object-oriented system form a hierarchy — the class hierarchy. A class may have one or more subclasses which are a specialization of the class. At the same time, a class may have one or more superclasses which are considered to be a generalization of the class.

  A subclass inherits all the attributes and methods of its superclass(es). In addition, it may define some additional attributes and methods, or redefine some of those which it inherits from its superclass(es). Object-oriented systems employ some conflict resolution mechanism if there is a conflict between the attributes and methods a class inherits from its various superclasses.

  The class hierarchy provides an important means for sharing of behavior among related objects. The principle of inheritance makes the task of the data base designer easier. In addition, the class hierarchy is also a more natural representation of real-world entities.

  In addition to the class hierarchy, the classes also participate in the class-composition hierarchy. Since an object may have several attributes, each of

which belongs to some class, an object may be thought of as being composed of several other objects. The class-composition hierarchy relates a class with the other classes that are the domains of the attributes of the class.

2.3 Overview of Transaction Management

In a typical data base environment, the users access data through the means of *transactions*. A transaction models a logical unit of work in the application environment. A transaction is also considered to be a unit of data base consistency, i.e., it takes the data base from one consistent state to another. Each transaction is assumed to execute in isolation without any interference from other transactions. The data base states arising out of partial execution of a transaction are considered to be inconsistent and are therefore not allowed to be observed by other transactions. However, to increase the throughput, most data base systems allow for concurrent execution of transactions. To ensure that the data base consistency is still maintained, these data bases employ some concurrency control protocols such as *two phase locking* or *time-stamp ordering*. The concurrency control protocols are designed to shield each individual transaction from the effects of the other concurrently executing transactions. The protocols enforce certain correctness criterion. In traditional data base systems, this correctness criterion is *serializability*.

Informally, a concurrent execution of transactions (also known as a schedule or history) is said to be serializable if it is *equivalent* to some serial execution of this set of transactions. Depending on how *equivalence* is defined, we get different notions of serializability. The two most popular notions are those of *View-serializability* and *Conflict-serializability*. However, checking whether a schedule is view-serializable is

an NP-complete problem. Therefore, most applications use conflict-serializability as their correctness criteria. In addition to enforcing the concurrency control mechanisms, the transaction management software also has to provide for recovery from system crashes. Transaction management for relational data bases has been extensively studied [8, 9, 10, 11, 12, 13, 14, 15].

2.4 Object-oriented Data Bases

Object-oriented data bases present some interesting problems when one considers the issue of transactions. The transactions in an object-oriented data base differ from those in a conventional data base system in many ways. We will now discuss some of the major differences, which include the duration of transactions, cooperation among transactions, the granularity of transaction management features, and the complications arising from the various object-oriented concepts like class and class-composition hierarchies.

2.4.1 Long-duration transactions

Object-oriented data bases are more suitable for applications like CAD, rather than for applications like Banking. The nature of transactions in a CAD system is very different from that in conventional systems. The typical duration of a transaction in a CAD environment may range from a few seconds to a few days. The concurrency control mechanisms like locking and time-stamps which are used in conventional data bases are inadequate for handling long-duration transactions. If locking is used for concurrency control, we have the problem of long-duration waits. Thus, a long-duration transaction may delay other transactions. Such long delays are un-

acceptable. If time-stamps are used, aborting a long-duration transaction will imply heavy penalties in terms of wasted computations and other critical resources. Thus there exists a need for a different mechanism to handle long-duration transactions.

2.4.2 Cooperation among transactions

Transactions in an object-oriented environment such as in CAD or MMDB systems are not necessarily designed to run in isolation. The long-duration transactions in applications such as above are generally designed so that they cooperate with each other. This means that partial results of a particular transaction may be seen by some other transaction with which it is cooperating. At the same time, these partial results may not be visible to some other transactions since they represent an *inconsistent* state of the data base. Thus *consistency* of the data base is now relative to a transaction. The data base state resulting from the partial execution of a transaction is acceptable to other transactions cooperating with it and at the same time unacceptable to some other transactions. The cooperating nature of the transactions also implies that serializability can no longer be the correctness criterion for a concurrent execution of a set of transactions. Even in the presence of non-serializable transaction executions (schedules), one may be able to obtain *correct* executions.

2.4.3 Granularity of transaction management features

In object-oriented systems, the objects are accessed via their identifiers. Thus the (logical) unit of access is in terms of objects. It seems natural that the unit of locking should also be objects, not classes. This contrasts with relational systems, where the (logical) unit of access could be relations or some field(s) within relations.

2.4.4 Impact of the class and class-composition hierarchies

The class and class-composition hierarchies interact with the handling of queries. The queries in an object-oriented system are much different from those in a relational system. Due to the nature of an object, a query against an object can become a recursive query against its components. Similarly, a query against a class can be treated as a query against all the subclasses of that class. This is because a class represents a generalization of all its subclasses. Thus, the class and the class-composition hierarchies make queries (and hence transactions) more complicated to handle. This means that locking a class object has implications in terms of locking an entire class hierarchy. If this is not done, serious problems may arise in maintaining the consistency of the data base.

2.4.5 Object-oriented data base systems - current literature

The published literature in the field of transaction management for object-oriented data base systems is lacking in many respects. Among these, the lack of a suitable transaction model is considered to be the most important. Past research in this area has taken the approach of simply extending the concepts developed in the relational framework to the object-oriented model. Unfortunately, such an extension tends to ignore the impact of the semantic richness of the object-oriented data model. Furthermore, the nature of transactions in applications such as CAD and MMDB systems is much different from that in the conventional data base systems. As noted earlier, these applications often involve long-duration cooperating transactions as opposed to the short-duration stand-alone ones in conventional data base systems. The complex interactions between transactions in such applications

also require alternate criterion for defining *correct* executions. The traditional notion of *serializability* is no longer appropriate in the presence of cooperating transactions.

While there is considerable literature on object-oriented data base systems [5, 7, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29], research on transaction management in object-oriented data bases is still in its infancy. Kim [5] discusses some of the issues related to transaction management. A transaction model is presented along with a locking protocol. However, long-duration cooperating transactions are not handled. In [17], the authors present a model of CAD transactions. This model discusses the notions of *check-in* and *check-out*. The authors do not discuss how the object-oriented model interacts with transaction management. Furthermore, long-duration transactions are simply treated as a series of short-duration transactions with serializability as the correctness criterion. Some work has been done in the area of cooperating transactions for conventional systems [30, 31]. However, object-oriented features are not discussed. Thus, there is a need for further research in the area of transaction management for object-oriented data bases.

2.5 Summary

In this chapter, we provided an overview of some object-oriented features including the concepts of class and class-composition hierarchies. The basics of transaction management and concurrency control were also discussed. The characteristic features of transactions in advanced applications in object-oriented data bases were discussed. Included among these were the duration of transaction and their cooperative nature. The lack of an appropriate transaction model for object-oriented data bases was identified. In the next chapter, we introduce our transaction model for object-oriented data bases.

3. TRANSACTION MODEL

3.1 Introduction

In chapters 1 and 2, we described the unique nature of transactions in advanced data base applications such as CAD and MMDB systems. We also identified the need for a new transaction model for object-oriented data base systems. In this chapter, we present a model based on cooperating long-duration transactions that is more appropriate for the applications arising in the object-oriented data base environment. This model is similar to the one presented in [31] and [30]; however, it is presented here in the context of object-oriented data bases and takes into consideration their inherent complexities.

We first provide a few definitions and introduce some notations in section 3.2. Section 3.3 presents our transaction model in detail. The model includes concepts of breakpoint and grouping specifications, a new definition for *conflicts* in object-oriented data base systems, a new correctness criterion, and the definitions of Object-Oriented Correct and Correctable Schedules. Section 3.4 provides a graph characterization for correctable schedules. A few concluding remarks are presented in section 3.5

3.2 Definitions and Notations

Let C be the set of all classes in the data base. Let PC be the set of all primitive classes in the data base ($PC \subset C$). Let the collection of all the objects in the data base be denoted by DB. Unless specified otherwise, we do not distinguish between objects which represent classes and objects which are *instances* of the class objects.

Definition 1:

A function $ClassOf$ is defined as follows:

$ClassOf : DB \mapsto C$

Intuitively, $ClassOf$ gives the domain of an object, i.e., the class object for which the object under consideration is an instance. Note that the class objects themselves are instances of the class **Class**.

Until now we have used an informal definition of an object. We will now define it more formally.

Definition 2:

Let object O be composed of instance variables iv_1, iv_2, ..., iv_m. An instance variable iv_j is a primitive instance variable if $ClassOf(iv_j) \in PC$. We assume that if iv_j is a primitive instance variable of object $O \in DB$, then no other $O' \in DB$ has a reference to iv_j. In other words, iv_j can only be accessed via object O. Let the operations (methods) be op_1, op_2, ..., op_n. Each operation op_i has a set (possibly empty) of parameters p_1, p_2, ..., p_r. We will use the terms *operations* and *methods* interchangeably.

Each operation op_i consists of a number of steps op_{i_1}, op_{i_2}, ..., op_{i_s}. The steps are totally ordered by the relation $<_{op_i(O)}$. Each of these steps is either a local step or a global step. A step is said to be a local step if it accesses a primitive instance

variable; otherwise, it is a global step. A global step consists of sending a message to an object and results in the invocation of an operation at the receiving object. We say that operation op_i on object O is the parent of operation op_j on object O' if op_j has been invoked by a global step of op_i sending a message to object O'. Operation op_j on object O' is referred to as a child of operation op_i on object O. The parent-child relationship can be generalized to the ancestor-descendent relationship via a transitive closure.

In the object-oriented model, we assume a *blocking* semantics, i.e., when a global step of $op_i(O)$ invokes operation $op_j(O')$, then $op_i(O)$ is suspended until $op_j(O')$ is completed. Thus, if $op_i(O)$ is the parent of $op_j(O')$, then $op_j(O')$ completes before $op_i(O)$.

One of the characteristics of an object-oriented system is encapsulation. However, some objects in the data base are denoted as system objects. The system objects are *globally visible* in the sense that any operation on any object can include a step which sends a message to one of these system objects.

3.2.1 Scope of an operation

Definition 3:

The *Scope* of an operation op_i on object O, $(Scope(op_i(O)))$, is defined to include the following objects:

1. All the instance variables of object O.

2. All the parameters $p_1, p_2, \ldots, p_r$ associated with the operation op_i on O.

3. All the system objects.

4. If O is a class object, the *Scope* of operation op_i on object O includes the direct subclasses of O and the classes $C_1, \ldots, C_m$, where each C_j is the domain of the instance variable iv_j respectively (i.e., $ClassOf(iv_j) = C_j$). In addition, all the objects which are instances of class O are also in the *Scope* of op_i on object O. We do not consider the instances of subclasses of O to be in the *Scope* of op_i on object O.

Intuitively, the *Scope* of an operation op_i on object O includes all the objects which the operation can directly manipulate (primitive instance variables of O) or the objects to which a direct message is sent by one of the steps of this operation.

An operation op_i on object O may directly access the primitive instance variables of object O through local steps and indirectly access other objects (and their primitive instance variables) through global steps. This is in contrast with the conventional data base systems in which an operation on any data base entity does not affect any other data base entity.

We can classify the effects of any operation (on an object) into four major categories:

- Query

  An operation may *read* the value of an object in the data base.

- Create

  An operation may create new instances of some class and thus add a new object to the data base.

- Modify

  An operation may modify the value of an existing object in the data base.

- Delete

  An operation may delete an existing object from the data base.

The class hierarchy, the class-composition hierarchy and the instance-of relationships lead to a cascade effect. For example, querying an object may in turn result in querying its instance variables and so on. Thus the effects of the above types of operations are not necessarily local. For each operation op_i on object O, we therefore define a QuerySet ($QS(op_i(O))$), a CreateSet ($CS(op_i(O))$), a ModifySet ($MS(op_i(O))$) and a DeleteSet ($DS(op_i(O))$).

Definition 4:

The QuerySet ($QS(op_i(O))$) is defined recursively as:

$QS(op_i(O)) = LocalQS(op_i(O)) \cup GlobalQS(op_i(O))$, where

1. $LocalQS =$

   - $\emptyset$, if none of the primitive iv_j's of O are *queried* by op_i.
   - $\{O\}$, otherwise.

2. $GlobalQS =$

   $\{O_{gq} \mid op_i$ sends a message to O_s to execute method op_j,
   where $O_s \in Scope(op_i(O))$, and $O_{gq} \in QS(op_j(O_s))$ $\}$.

We can analogously define the CreateSet, ModifySet and the DeleteSet of an operation op_i on object O.

Definition 5:

Let the UpdateSet (US) be defined as:

$US(op_i(O)) = (CS(op_i(O)) \cup MS(op_i(O)) \cup DS(op_i(O)))$.

We now give a definition for *conflicts* in an object-oriented environment.

3.2.2 Conflicts

Definition 6:

Two operations op_i on object O and op_j on object O' are said to *conflict* if either:

1. $US(op_i(O)) \cap US(op_j(O')) \neq \emptyset$, or
2. $QS(op_i(O)) \cap US(op_j(O')) \neq \emptyset$, or
3. $US(op_i(O)) \cap QS(op_j(O')) \neq \emptyset$

The above notion of conflict differs from the notion of conflict in conventional data bases. In conventional data bases, two operations are said to conflict if they are on the *same* data element and at least one of them is a *write*. However, in the object-oriented case, two operations on two different objects may also conflict. This is because of the semantic richness of the object-oriented data model which is much different from the simpler *flat* nature of the relational model.

3.3 Modeling Cooperating Transactions

The need for alternate correctness criterion for concurrent transaction execution in an object-oriented data base has been demonstrated. The concept of multilevel atomicity has been employed in the past [31, 30] for cooperating transactions in conventional data base environments. [16] addresses some of the issues for transaction synchronization in *object-bases*; however, neither does it address the complications arising out of the class hierarchy, the class-composition hierarchy and inheritance in object-oriented data bases, nor does it deal with cooperating transactions. We now present a transaction model for long-duration cooperating transactions in an object-oriented environment.

We will consider the (user) transactions as executions of methods or operations of a special data base object, the Data Base User-Interface Object (DBIO). Thus the transactions will be treated like any other operation or method in the system. In conventional data bases, an operation on a data entity is considered to be an atomic action. For instance, *read* or *write* of any data entity is an atomic action. However, the operations (on objects) in object-oriented data bases cannot be assumed to be atomic. Each operation may consist of several local and global steps and may result in invocation of operations at other objects. To increase concurrency, we should allow several operations to be active at the same object. However, to maintain correctness, we need concurrency control mechanisms. Like transactions in such an environment, some of these operations may cooperate with other simultaneously active ones and observe each others partial results. Thus we may have cooperating operations within the same object.

3.3.1 Grouping specification

For each object O in the data base, let $OpTypes(O)$ be the set of types of methods or operations allowable on object O.

Let $OpTypes(O) = \{OpType_1, OpType_2, \ldots, OpType_p\}$. $OpTypes(O)$ provides a mechanism for classifying the operations op_1, op_2, ..., op_n of object O. For each operation op_i, there is an associated operation type $OpType_j \in OpTypes(O)$.

Definition 7:

Let CG_O be a k-level grouping specification for the set $OpTypes(O)$. CG_O is defined as follows:

- $CG_O(1)$ consists of only one set - $OpTypes$.

- $CG_O(k)$ consists of singleton sets, one for each operation type in the set $OpTypes$.

- Each $CG_O(i)$ is a refinement of $CG_O(i-1)$, i.e., for any set $cgs \in CG_O(i)$, there is a set $cgs' \in CG_O(i-1)$ such that $cgs \subseteq cgs'$.

Definition 8:

If op_i and op_j are two operations on O, and $OpType_i$ and $OpType_j$ are respectively their associated types, we define $Level(op_i(O), op_j(O)) = l$, where $CG_O(l)$ contains $OpType_i$ and $OpType_j$ in the same set, and for all $m > l$, $OpType_i$ and $OpType_j$ are not contained in the same set of $CG_O(m)$.

The grouping specification is based on the semantics of object O and its operations. The grouping specification indicates the cooperating nature of the different operation types. In particular, $OpTypes(DBIO)$ represents the types of user transactions which can occur in the application data base and CG_{DBIO} provides a specification describing the cooperating nature of the user transactions in the data base.

As an example, consider an application environment which deals with publishing new books. Consider an object called *BookObject*. It has the following operations defined:

1. AddText
2. AddFigures
3. EditChapters
4. ReviewBook

For simplicity, let the types of these operations be the same as their names. Consider the following 4-level grouping specification for the operations on *BookObject*:

- $CG_{BookObject}(1) =$
  $\{\{AddText, AddFigures, EditChapters, ReviewBook\}\}$.
- $CG_{BookObject}(2) =$
  $\{\{AddText, AddFigures, EditChapters\}, \{ReviewBook\}\}$.
- $CG_{BookObject}(3) =$
  $\{\{AddText, AddFigures\}, \{EditChapters\}, \{ReviewBook\}\}$.
- $CG_{BookObject}(4) =$
  $\{\{AddText\}, \{AddFigures\}, \{EditChapters\}, \{ReviewBook\}\}$.

The grouping specification indicates the kind of cooperation that is expected when a book is being created. The *AddText* and the *AddFigures* operations will be invoked to write new material with some illustrations. Every completed chapter may be examined by the editor via the *EditChapters* operation. Finally, a complete book may be reviewed by referees using the *ReviewBook* operation. The grouping specification indicates that the *ReviewBook* operation may not see the partial results of the book creation process. However, the *EditChapters* operation is allowed to see some of the partial results, i.e., whenever a chapter is completed. The *AddText* and the *AddFigures* are expected to cooperate to a greater degree and work towards creation of new chapters and finally the complete book.

The grouping specification for operations on an object indicates which operations may cooperate. Cooperating operations may see the intermediate results of each

other. However, we may want to restrict the partial results which are visible among these operations. This can be achieved by providing a k-level breakpoint specification.

3.3.2 Breakpoint specification

Definition 9:

For each operation op_i on object O, the k-level breakpoint specification $B_{op_i(O)}$ is defined as follows:

- $B_{op_i(O)}(1)$ consists of only one set - $\{op_{i_1}, op_{i_2}, \ldots, op_{i_s}\}$.
- $B_{op_i(O)}(k)$ consists of singleton sets, one for each step of operation op_i on object O.
- Each $B_{op_i(O)}(i)$ is a refinement of $B_{op_i(O)}(i-1)$.
- If $b_l \in B_{op_i(O)}(k)$, $op_{i_j} \in b_l$, $op_{i_n} \in b_l$, and there exists op_{i_m} such that $op_{i_j} <_{op_i(O)} op_{i_m} <_{op_i(O)} op_{i_n}$, then $op_{i_m} \in b_l$.
  (i.e., if two steps of an operation are in the same equivalence class, then all intermediate steps are also in the same equivalence class.)

To illustrate the idea of breakpoint specification, we will once again consider the *BookObject* example. For the sake of this example, assume that the book will consist of 3 chapters with 3 sections in each chapter. Also assume that each section will have one illustration. Consider the following 4-level breakpoint specification for the operations on *BookObject* (for ease of reading, we will drop the object descriptor):

1. Operation *AddText*

- $B_{AddText}(1) =$
  $\{\{t_{11}, t_{12}, t_{13}, t_{21}, t_{22}, t_{23}, t_{31}, t_{32}, t_{33}\}\}$.
- $B_{AddText}(2) =$
  $\{\{t_{11}, t_{12}, t_{13}\}, \{t_{21}, t_{22}, t_{23}\}, \{t_{31}, t_{32}, t_{33}\}\}$.
- $B_{AddText}(3) =$
  $\{\{t_{11}\}, \{t_{12}\}, \{t_{13}\}, \{t_{21}\}, \{t_{22}\}, \{t_{23}\}, \{t_{31}\}, \{t_{32}\}, \{t_{33}\}\}$.
- $B_{AddText}(4) =$
  $\{\{t_{11}\}, \{t_{12}\}, \{t_{13}\}, \{t_{21}\}, \{t_{22}\}, \{t_{23}\}, \{t_{31}\}, \{t_{32}\}, \{t_{33}\}\}$.

2. Operation *AddFigures*

- $B_{AddFigures}(1) =$
  $\{\{f_{11}, f_{12}, f_{13}, f_{21}, f_{22}, f_{23}, f_{31}, f_{32}, f_{33}\}\}$.
- $B_{AddFigures}(2) =$
  $\{\{f_{11}, f_{12}, f_{13}\}, \{f_{21}, f_{22}, f_{23}\}, \{f_{31}, f_{32}, f_{33}\}\}$.
- $B_{AddFigures}(3) =$
  $\{\{f_{11}\}, \{f_{12}\}, \{f_{13}\}, \{f_{21}\}, \{f_{22}\}, \{f_{23}\}, \{f_{31}\}, \{f_{32}\}, \{f_{33}\}\}$.
- $B_{AddFigures}(4) =$
  $\{\{f_{11}\}, \{f_{12}\}, \{f_{13}\}, \{f_{21}\}, \{f_{22}\}, \{f_{23}\}, \{f_{31}\}, \{f_{32}\}, \{f_{33}\}\}$.

3. Operation *EditChapters*

- $B_{EditChapters}(1) = \{\{c_1, c_2, c_3\}\}$.
- $B_{EditChapters}(2) = \{\{c_1\}, \{c_2\}, \{c_3\}\}$.
- $B_{EditChapters}(3) = \{\{c_1\}, \{c_2\}, \{c_3\}\}$.

- $B_{EditChapters}(4) = \{\{e_1\}, \{e_2\}, \{e_3\}\}$.

4. Operation *ReviewBook*

- $B_{ReviewBook}(1) = \{r_1\}$.
- $B_{ReviewBook}(2) = \{r_1\}$.
- $B_{ReviewBook}(3) = \{r_1\}$.
- $B_{ReviewBook}(4) = \{r_1\}$.

The *AddText* operation has one step for each of the sections in the book. For example, step t_{23} corresponds to the third section of the second chapter. Similarly, the steps of the *AddFigures* operation correspond to the illustration in each section. The three steps in the *EditChapters* operation correspond to editing each chapter. Finally, the *ReviewBook* operation has only one step. According to the breakpoint specification (in combination with the grouping specification), the text for each section and the figure may be interleaved. However, the *EditChapters* operation can see only the results at the end of each chapter, but not each section individually. The *ReviewBook* operation is restricted to seeing only the end results and cannot see individual sections or chapters as they are completed. Thus, the grouping specification and the breakpoint specification together describe the nature of cooperation among the operations on an object.

3.3.3 Object-oriented Correct Schedules (OOCS)

The *state* s of the data base at any instant is the collection of the values of the objects in the data base at that instant. When an operation is invoked on an object,

it may lead to a change in the state of that object and some other objects in the data base. Thus an operation on an object may result in a *state transition*. We say operation $op_i(O)$ is *defined* on state s if its execution does not result in an error condition. This notion is also extended to a sequence of operations. The function *NextState* denotes the new state resulting from an operation.

Definition 10:

If operation $op_i(O)$ acts on the state s_1 of the data base and the resulting state is s_2, then $s_2 = NextState(op_i(O), s_1)$. We can generalize the $NextState$ function to denote the state resulting from a sequence of operations $Ops = \{op_1(O_1), op_2(O_2), \ldots, op_n(O_n)\}$. i.e., if $s_2 = NextState(op_1(O_1), s_1)$, $s_3 = NextState(op_2(O_2), s_2)$, $\ldots$, $s_{n+1} = NextState(op_n(O_n), s_n)$, then we can also denote it as:

$s_{n+1} = NextState(Ops, s_1)$.

Note that if two operations $op_i(O)$ and $op_j(O')$ do not conflict, then $NextState(\{op_i(O), op_j(O')\}, s) = NextState(\{op_j(O'), op_i(O)\}, s)$, for any state s such that both $\{op_i(O), op_j(O')\}$ and $\{op_j(O'), op_i(O)\}$ are defined on state s.

Lemma 1:

If $Ops_1 = \{op_1(O_1), op_2(O_2), \ldots, op_n(O_n), op(O)\}$, $Ops_2 = \{op(O), op_1(O_1), op_2(O_2), \ldots, op_n(O_n)\}$, both Ops_1 and Ops_2 are defined on state s, and no other operation in Ops_1 conflicts with $op(O)$, then $NextState(Ops_1, s) = NextState(Ops_2, s)$.

Proof:

We will use induction to prove Lemma 1.

For $n = 1$, $NextState(\{op_1(O_1), op(O)\}, s) = NextState(\{op(O), op_1(O_1)\}, s)$ is trivially true.

For the induction step, assume that the lemma holds for a sequence of $n - 1$ opera-

tions, i.e., $NextState(Ops'_1, s) = NextState(Ops'_2, s)$, where

$Ops'_1 = \{op_1(O_1), op_2(O_2), \ldots, op_{n-1}(O_{n-1}), op(O)\}$ and

$Ops'_2 = \{op(O), op_1(O_1), op_2(O_2), \ldots, op_{n-1}(O_{n-1})\}$.

Let $NextState(\{op_1(O_1), op_2(O_2), \ldots, op_{n-1}(O_{n-1})\}, s) = s'$

Now consider $NextState(\{\{Ops'_1\}, op_n(O_n)\}, s)$. We have

$NextState(\{\{Ops'_1\}, op_n(O_n)\}, s)$

$= NextState(\{op_1(O_1), op_2(O_2), \ldots, op_{n-1}(O_{n-1}), op(O), op_n(O_n)\}, s)$

$= NextState(\{op(O), op_1(O_1), op_2(O_2), \ldots, op_{n-1}(O_{n-1}), op_n(O_n)\}, s)$ (by induction hypothesis)

$= NextState(Ops_2, s)$.

Also, $NextState(\{\{Ops'_2\}, op_n(O_n)\}, s)$

$= NextState(\{op(O), op_1(O_1), op_2(O_2), \ldots, op_{n-1}(O_{n-1}), op_n(O_n)\}, s)$

$= NextState(\{op_1(O_1), op_2(O_2), \ldots, op_{n-1}(O_{n-1}), op(O), op_n(O_n)\}, s)$ (by induction hypothesis).

$= NextState(\{op(O), op_n(O_n)\}, s') = NextState(\{op_n(O_n), op(O)\}, s')$ (since $op_n(O_n)$ and $op(O)$ do not conflict)

$= NextState(Ops1, s)$

Thus we have $NextState(Ops_1, s) = NextState(Ops_2, s)$. □

Lemma 2:

If Ops and Ops' are two sequences of operations such that both are defined on state s, they contain the same operations, and all pairs of conflicting operations are similarly ordered in both the sequences (i.e., if $op_i(O_i)$ and $op_j(O_j)$ conflict and $op_i(O_i) <_{Ops} op_j(O_j)$, then $op_i(O_i) <_{Ops'} op_j(O_j)$), then $NextState(Ops, s) = NextState(Ops', s)$

Proof:

We prove Lemma 2 by using induction and Lemma 1.

Let $Ops = \{op_1(O_1), op_2(O_2), \ldots, op_n(O_n)\}$ and $Ops' = \{op'_1(O'_1), op'_2(O'_2), \ldots, op'_n(O'_n)\}$, such that they contain the same operations and all pairs of operations are similarly ordered.

For $n = 1$, the proof is trivial. For the induction step, assume that the lemma holds for a sequence of $(n-1)$ operations. Now consider sequences of n operations. Let $op_n(O_n) \in Ops$ be the same as $op'_k(O'_k) \in Ops'$. Since the conflicting operations are similarly ordered in both the sequences, $op'_k(O'_k)$ commutes with each $op'_j(O'_j)$, for $k < j < n$.

Let $Ops'' = \{op''_1(O''_1), op''_2(O''_2), \ldots, op''_n(O''_n)\}$, such that

$$op''_i(O''_i) = \begin{cases} op'_i(O'_i), \text{ for } 1 \leq i < k, \\ op'_{i+1}(O'_{i+1}), \text{ for } k \leq i < n. \\ op'_k(O'_k), \text{ for } i = n. \end{cases}$$

Let Ops_{n-1} and Ops''_{n-1} be subsequences containing the first $(n-1)$ operations of Ops and Ops'' respectively. Note that both subsequences have the same set of operations, and the conflicting operations are ordered similarly. By induction hypothesis, $NextState(Ops_{n-1}, s) = NextState(Ops''_{n-1}, s)$, where s is the initial state.

Let $NextState(Ops''_{k-1}, s) = s'$ Note that $NextState(Ops'_{k-1}, s) = s'$, by definition of Ops''. Using Lemma 1, on the remainder of operations, we have $NextState(Ops', s) = NextState(Ops'', s)$. Since $op''_n(O''_n) \in Ops''$ is the same as $op_n(O_n) \in Ops$, using the induction hypothesis, we have $NextState(Ops, s) = NextState(Ops'', s)$. Thus we have $NextState(Ops, s) = NextState(Ops', s)$. □

To increase the throughput of the data base system, several operations may be concurrently active across the data base. Some of these operations may be on the same object. The concurrent execution of multiple operations may lead to arbitrary interleaving of operations (and their steps). Such an interleaving can be referred to as an *object-oriented schedule (OOS).* Since the *user transactions* are just operations on DBIO, more formally we define an OOS *Sch* on DBIO as a 2-tuple $(S, <_{Sch})$, where S is the set of all steps (local as well as global) resulting from the invocation of operations on DBIO and $<_{Sch}$ is a partial order on the steps in S. Note that the global steps in S will result in invocation of operations at other objects and the steps executed as a result of this are also included in *Sch*.

Let the *user transactions* invoke operations

$T = \{t_{op_1}(DBIO), t_{op_2}(DBIO), \ldots . t_{op_n}(DBIO)\}$.

Note that some of these may be separate invocations of the same operation on DBIO, i.e., different instances of operation executions on DBIO. Let $T' = \{t'(O') \mid t'(O')$ is a descendent of operation $t(DBIO) \in T\}$.

Definition 11:

A schedule *Sch* for T on DBIO is said to be an *Object-Oriented Correct Schedule* (OOCS) if

1. *Sch* consists only of steps from operations in T' and each of these steps occurs exactly once in *Sch*.

2. $<_{Sch} \supseteq <_{t'(O')}$, for each operation $t'(O') \in T'$.
   In other words, for each operation occurring in it, the schedule *Sch* does not violate the ordering of steps which is specified for that operation.

3. If $t'(O) \in T'$, $t''(O) \in T'$,

   $Level(t'(O), t''(O)) = l$,

   $t'(O) = \{t'_1(O), t'_2(O), \ldots t'_m(O)\}$,

   $t''(O) = \{t''_1(O), t''_2(O), \ldots t''_n(O)\}$,

   $t'_p(O), t'_r(O) \in b'_l, b'_l \in B_{t'(O)}(l)$,

   $t''_q(O) \in b''_l, b''_l \in B_{t''(O)}(l)$,

   $t'_p(O) <_{t'(O)} t'_r(O)$, $t'_p(O) <_{Sch} t''_q(O)$,

   then

   $t'_r(O) <_{Sch} t''_q(O)$.

   This implies that an Object-Oriented Correct Schedule will have to guarantee that the breakpoint specifications for operations are observed properly, i.e., other cooperating operations may not see partial results other than those described by the breakpoint specification.

3.3.4 Object-oriented Correctable Schedules (OOCLS)

Definition 12:

An OOS Sch_1 is said to be *equivalent* to another OOS Sch_2 if

1. Both are defined over the same set of operations (say T).

2. If $t'(O'), t''(O'') \in T$,

   $t'_p(O')$ is a step of $t'(O')$,

   $t''_q(O'')$ is a step of $t''(O'')$,

   $t'_p(O') <_{Sch_1} t''_q(O'')$, $t'_p(O')$ conflicts with $t''_q(O'')$,

   then

$t'_p(O') <_{Sch_2} t''_q(O'')$.

In other words, both the schedules Sch_1 and Sch_2 order the conflicting operations similarly.

Theorem 1:

If Sch_1 and Sch_2 are OOS's which are defined on state s of data base, and Sch_1 is equivalent to Sch_2, then $NextState(Sch_1, s) = NextState(Sch_2, s)$.

Proof:

Since Sch_1 is equivalent to Sch_2, both the schedules contain the same set of operations and conflicting operations are ordered similarly by both the schedules. Sch_1 and Sch_2 satisfy the conditions for Lemma 2, and therefore we have $NextState(Sch_1, s) = NextState(Sch_2, s)$. □

Definition 13:

An OOS Sch is said to be *Object-Oriented Correctable Schedule* (OOCLS) if it is equivalent to some OOCS Sch'.

The notions of intra-object and inter-object synchronization for object bases were introduced in [16]. In our transaction model, the synchronization described above can be treated as being intra-object. The grouping specification for operations on an object and the breakpoint specification for each operation exploit the semantic information about the objects. Thus concurrency control is carried out at each individual object. In [16], inter-object synchronization is also needed to guarantee serializability of transaction executions. In our model, the transactions are no different from any other operation in the data base. Therefore the intra-object synchronization being provided for the DBIO (by the grouping and breakpoint specification) serves the purpose of inter-object synchronization to guarantee object-oriented correctable

schedules. Thus in our transaction model, no explicit inter-object synchronization is necessary.

3.4 A Graph Characterization for Correctable Schedules

The literature on concurrency control which deals with serializability as the correctness criterion often provides for a *no-cycles* graph characterization for the serializability problem. However, for the transaction model presented in this chapter wherein the transactions (or rather, the operations on DBIO) are expected to cooperate and observe each others' partial results, such a *no-cycles* characterization is not possible. However, we offer a graphical characterization which is based on the grouping and the breakpoint specifications.

Consider a schedule $Sch = (S, <_{Sch})$ over the set of operations T. Let T' be as defined for definition 11. Consider two operations op_i and op_j in T' which are on the same object, say O. Let $Level(op_i(O), op_j(O)) = l$. Let $B_{op_i(O)}(l) = \{b_{i1}, b_{i2}, \ldots, b_{ip}\}$ and $B_{op_j(O)}(l) = \{b_{j1}, b_{j2}, \ldots, b_{jq}\}$.

Definition 14:

We define an Operations Interaction Graph ($OIG_{Sch}(op_i(O), op_j(O))$) between operations $op_i(O)$ and $op_j(O)$ as follows: $OIG_{Sch}(op_i(O), op_j(O)) = (V, E)$, where

- V is the set of nodes.

  The set of nodes V consists of the elements of $B_{op_i(O)}(l)$ and $B_{op_j(O)}(l)$.

- E is the set of edges

  The set of edges E consists of the following:

  1. There is an edge from b_{ik} to $b_{i(k+1)}$ for $1 \leq k \leq (p-1)$, and from b_{jl} to

$b_{j(l+1)}$ for $1 \leq l \leq (q-1)$.

2. There is an edge from b_{im} to b_{jn} whenever a step $op_{i_s} \in b_{im}$ conflicts with a step $op_{j_r} \in b_{jn}$ and $op_{i_s} <_{Sch} op_{j_r}$.

3. There is an edge from b_{jn} to b_{im} whenever a step $op_{j_r} \in b_{jn}$ conflicts with a step $op_{i_s} \in b_{im}$ and $op_{j_r} <_{Sch} op_{i_s}$.

Note that any cycle in the OIG implies that the interleaving of steps of the operations in the OIG violated the breakpoint specification for those operations. The absence of cycles indicates that the breakpoint specification is not violated.

Theorem 2:

Let *Sch* be an OOS. If for each object O which receives a message in T' (as defined for definition 11), the OIG for each pair of operations on O is acyclic, then *Sch* is OOCLS.

Proof:

For convenience, let G denote the set of OIG's obtained by considering all pairs of operations on all objects receiving a message in T'. For each O which receives a message in T', let G_O denote the set of OIG's which involve object O ($G_O \subseteq G$). Since each OIG in G_O is acyclic, for each O, it satisfies the breakpoint specification for each pair of operations on O and hence *Sch* is OOCLS. □

3.5 Summary

The need for a new transaction model for object-oriented data base systems was identified in the previous chapters. In this chapter we presented our transaction model. The transaction model was characterized by the following:

- Support for long-duration cooperating transactions
- Correctness criterion other than serializability
- Support for object-oriented features including class and class-composition hierarchies
- Support for exploiting semantic knowledge provided by application programs

The transaction model presented in this chapter is a major improvement over previous work in this area which primarily involved a mere extension of transaction management features from relational systems to object-oriented systems. Such an extension failed to address the unique needs of object-oriented systems. Our transaction model defines correct and correctable executions. In the next chapter, we describe a concurrency control protocol that can be used with our model. We will also prove the correctness of the protocol.

4. CONCURRENCY CONTROL PROTOCOL

4.1 Introduction

Concurrency control is an important aspect of transaction management in a data base system. In conventional data bases, transactions are considered to be independent atomic entities which execute in isolation and are insulated from the effects of other transactions. However, to maximize the system utilization, several transactions are scheduled for concurrent execution. Concurrency control mechanisms are used to ensure correct execution of this set of transactions. Concurrency control issues are more complicated in object-oriented data bases with cooperating long-duration transactions.

In an environment of cooperating long-duration transactions, concurrency control has to allow for user-defined interleavings and at the same time prevent incorrect interleavings. Furthermore, the object-oriented features such as the class and class-composition hierarchies also affect the concurrency control mechanisms. The semantic richness of the object-oriented data model combined with run-time features such as dynamic binding make concurrency control issues non-trivial. A new transaction model for object-oriented data base systems was discussed in the previous chapter. The model incorporated the above-mentioned features and also defined object-oriented correct and correctable schedules. In this chapter, we propose a new

concurrency control mechanism that can be employed in such an object-oriented data base system. The proposed concurrency control mechanism satisfies the requirements of correctable schedules.

Section 4.2 discusses the existing concurrency control protocols and their drawbacks. Section 4.3 introduces our concurrency control protocol. An example to illustrate the protocol is provided in section 4.4. The proof of correctness for the protocol is provided in section 4.5. Section 4.6 compares our protocol with the existing literature on concurrency control and section 4.7 provides a few concluding remarks.

4.2 Background

Unlike conventional data bases, research in the area of concurrency control for object-oriented data bases has been rather limited. The existing literature on concurrency control fails to meet the unique requirements of object-oriented data base systems. The lack of a universally accepted data model is one of the contributing factors for this paucity of research. Furthermore, the nature of transactions in advanced applications, such as computer aided design (CAD) and multimedia data bases (MMDB), is much different. These applications are characterized by long-duration cooperating transactions with generalized operations. Since the transactions are allowed to observe the intermediate results of other transactions, serializability can no longer be the correctness criterion of choice.

4.2.1 Requirements for concurrency control protocols

Keeping in mind the unique nature of transactions occurring in object-oriented data bases used by advanced applications, concurrency control protocols for such

data bases need to satisfy the following requirements:

- Handle long-duration cooperating transactions (LO)
- Enforce correctness criterion other than serializability (CO)
- Deal with object-oriented features (OO)
- Allow generalized operations (i.e., other than read and write) (GO)

4.2.2 Current literature

Some existing concurrency control mechanisms which are closest to the requirements stated above are described in [5, 16, 31, 32, 30]. However, they fail to meet one or more of these requirements. In [5], Kim discusses some of the transaction management issues in object-oriented data bases. However, the discussion does not include long-duration cooperating transactions. Serializability is the correctness criterion. Hadzilacos and Hadzilacos [16] discuss transaction management in object bases. Again, the transactions are short-duration non-interacting ones and the correctness criterion is serializability. Lynch [31], Farrag and Ozsu [32], and Garcia-Molina [30] describe transaction management for long-duration cooperating transactions. Correctness criterion other than serializability is used. However, their work is in the context of conventional data bases and therefore object-oriented features are not discussed. Generalized operations, i.e., operations other than read and write, are not considered in [5, 16, 31, 32, 30]. Table 4.1 summarizes the shortcomings mentioned above.

Table 4.1 clearly illustrates the need for a new concurrency control protocol that is capable of handling the kinds of transactions arising in advanced data base

Table 4.1: Comparison of existing literature

	LO	CO	OO	GO
[5]	No	No	Yes	No
[16]	No	No	Yes	No
[31]	Yes	Yes	No	No
[32]	Yes	Yes	No	No
[30]	Yes	Yes	No	No

applications. The next section introduces our concurrency control protocol. The protocol described in this chapter assumes the underlying object-oriented transaction model described in the previous chapter [33].

4.3 New Protocol

In the previous section we outlined the characteristics of transactions in advanced applications that use object-oriented data base systems. We also noted the lack of suitable concurrency control protocols to handle such transactions. In this section, we present a concurrency control protocol for an object-oriented data base system. The protocol attempts to address the requirements mentioned in the previous section.

- Support for long-duration cooperating transactions.
- Provide correctness criterion other than serializability.
- Support for object-oriented features such as the class and the class-composition hierarchies. Late binding is another feature of object-oriented systems which further complicates the concurrency control mechanism.
- Support for generalized operations other than read and write.

- Support for exploiting semantic richness of the object-oriented data model.

4.3.1 Overview of the protocol

If the objects that are accessed by different transactions are known a-priori, the task of the concurrency control mechanisms is relatively easy. The a-priori information facilitates a static determination of the conflicting operations and hence a strategy to regulate the interleaving of operations can be formulated. However, the late binding feature of object-oriented systems makes it difficult to determine a-priori the accessed objects. In the absence of such knowledge, one approach is to block some of the transactions until the accessed objects are known. However, since these are long-duration transactions, such blocking may lead to unacceptable long-duration waits.

Our protocol takes an *optimistic* approach when a-priori knowledge is not available. When a protocol uses the optimistic approach, incorrect executions are detected once all the accessed objects are known. In the event of incorrect interleavings, one or more transactions (operations) have to be either aborted or rolled back to rectify the incorrect execution. Ordinarily, this is not a bad approach. However, in advanced applications which we consider, the transactions are long-duration ones. The roll-back or abortion of such long-duration transactions is very expensive in terms of the system resources that are wasted and the time that is lost in useless computations.

An obvious solution is to limit the amount of roll-back. We use the idea of checkpoints to limit the amount of roll-back. In the event of an abort or roll-back, the effects up to the previous checkpoint have to be undone. This approach minimizes the waste of resources and at the same time reduces long-duration waits.

4.3.2 Checkpoint specification

The idea of checkpoints is used to minimize the amount of rollback in case of transaction aborts. These checkpoints can be specified by the user. Checkpoints are associated with operations on objects and may be specified as a step of the operation. We do not expect to have a checkpoint specification for each object in the system. Instead, the user may specify checkpoints at some points in the operations on some of the objects, such that each point represents a logical unit of work. The idea behind checkpoints is to provide a common point of reference at which the data base may be assumed to be in an acceptable state. The checkpoint serves as a synchronization mechanism as far as the coherency of the data base is concerned. Each user transaction can potentially depend on the actions of other transactions. Therefore, a checkpoint in a transaction by itself is meaningful only if all the other active operations also agree that the data base state at that point in time represents a consistent state. In that sense, a checkpoint acts as a rendezvous point where all active transactions in the system commit their (partial) work done up to that point.

The data base application designer is assumed to have considerable knowledge regarding the semantics of the operations of the data base. The semantic knowledge can be used to introduce checkpoints in transactions at points which correspond to completion of logical units of work. In conventional data bases with short-duration transactions, the entire transaction is a logical unit of work. However, we cannot treat the entire transaction in advanced applications as a logical unit of work. This is where the idea of checkpoints is useful.

4.3.3 Protocol description

The concurrency control protocol described in this section requires some information to be stored at each object. In some cases, it may be sufficient to store the information only at the class objects and not with every instance object. However, a discussion of these efficiency issues is beyond the scope of this research. Although we stated in the previous chapter that there is no need for inter-object concurrency control, the protocol described here requires some amount of inter-object concurrency control. We can do away with inter-object concurrency control, but at the expense of effective utilization of the system. To increase throughput, we allow for interleavings which although not specified by the grouping and breakpoint specifications, yield correctable executions. However, this requires us to have some degree of inter-object concurrency control.

Each object in the system stores the following information:

- Breakpoint/Interleaving Table
- Conflict Table
- Dependency Lists and Dependency Graph

4.3.3.1 Breakpoint/Interleaving Table (static) The Breakpoint/Interleaving Table is derived from the breakpoint and grouping specifications for the object and has the form shown in Table 4.2.

The entry in the ith row and jth column of Table 4.2, i.e., $E_{BI}(i,j)$, is the set of breakpoints of $op_j(O)$ at which $op_i(O)$ can interleave. The static information provided by the Breakpoint/Interleaving Table is combined with the run-time infor-

Table 4.2: Breakpoint/Interleaving Table

Operation Requested	Operations executing					
	op_1	op_2	...	op_j	...	op_n
op_1						
op_2						
⋮						
op_i				$E_{BI}(i,j)$		
⋮						
op_n						

Table 4.3: Conflict Table

Operation Requested	Operations executing					
	op_1	op_2	...	op_j	...	op_n
op_1						
op_2						
⋮						
op_i				$E_{Con}(i,j)$		
⋮						
op_n						

mation on the progress of the executing operations i.e., what step each operation is currently executing.

4.3.3.2 Conflict Table The Conflict Table is similar to the Breakpoint/Interleaving Table and is shown in Table 4.3.

The entry in the ith row and jth column of Table 4.3, i.e., $E_{Con}(i,j)$, is:

- *Yes*, if $op_i(O)$ conflicts with $op_j(O)$.

- *No*, if $op_i(O)$ does not conflict with $op_j(O)$.
- *Unknown*, if it is not known a-priori if $op_i(O)$ conflicts with $op_j(O)$.

The *Unknown* entry is a result of the late binding feature of object-oriented systems wherein the identity of some of the objects accessed may not be known until run-time. Therefore, it is not possible to determine a-priori whether the two operations conflict. The entries *Yes* and *No* are used when it is possible to statically determine whether two operations will conflict (or not). Note that the notion of conflict used in constructing the Conflict Table is the same as described in the previous chapter [33].

4.3.3.3 Dependency Lists and Dependency Graph As described in the previous chapter, the user transactions are invoked on a special object called *DBIO* (Data Base Interface Object). If $op(O)$ is an operation, let $Tr(op(O))$ denote the user transaction (i.e., the operation on *DBIO* which is the ancestor of $op(O)$). A dependency list consists of zero or more dependency items. Each dependency item (or sometimes referred to simply as a dependency) is of the form $\langle T_i, T_j \rangle$ where T_i and T_j are user transactions such that T_i follows and conflicts with T_j.

We define three kinds of dependency lists:

- Local Dependency List

  Each object O maintains a local dependency list, denoted by $DS_L(O)$. This list contains dependency items generated locally at object O. Initially, this list is empty and dependency items are added as and when the user transactions (directly or indirectly) access local information in a conflicting fashion.

- Inherited Dependency List

  Each object O also maintains an inherited dependency list, denoted by $DS_I(O)$. This list is also initially empty. When an operation $op(O)$ is invoked at O as a result of a message from $op_i(O_i)$ from the calling object, the calling object also sends its dependency set $DS(O_i)$ (defined later). The dependency items from $DS(O_i)$ are added to $DS_I(O)$. The inherited dependency list can be considered as a form of forward propagation of dependencies from the calling operation (object) to the called operation (object).

- Acquired Dependency List

  The acquired dependency list, denoted by $DS_R(O)$ is augmented when an operation $op_r(O_r)$ called as a result of a global step of an operation $op(O)$ of O is completed. Along with the results of the called operation, the called object also sends its dependency set $DS(O_r)$. The acquired dependency list can be considered as a form of backward propagation of dependencies from the called object to the calling object.

The dependency set at O is defined as

$$DS(O) = DS_L(O) \cup DS_I(O) \cup DS_R(O).$$

The dependency graph $DG(O)$ at each object O is a graphical representation of the dependency set $DS(O)$. The nodes of this graph consist of user transactions. For each dependency item $\langle T_i, T_j \rangle$, there is an edge $T_i \rightarrow T_j$ in DG.

In the transaction model which we have adopted, the transactions may observe each others' intermediate results which are allowed by the breakpoint and the grouping specifications. In conventional data base systems, the dependency items are never deleted from the dependency lists. However, in our case, some of the dependency

items may be deleted as and when the operations reach their breakpoints. Thus the dependency graph has a slightly different meaning in our context. The presence of cycles still indicates incorrect execution. However, it is important to note that dependencies are added as well as deleted over time. The addition and deletion of dependencies are described later.

4.3.3.4 Status of user transactions at objects Each object O in the system stores the *status* of every user transaction in the system. The status of a user transaction (i.e., an operation on $DBIO$) can have one of the following values:

- NeverActivated

  Any user transaction which does not directly or indirectly access O has the status NeverActivated at O. This is equivalent to not having information of that user transaction at O.

- Completed

  A user transaction has the status Completed at O, if the operation invoked by it at O (say $op(O)$) has completed executing all its steps.

- CheckPointed

  A user transaction is said to have the status CheckPointed at O if it has not performed any step after a checkpoint.

- BlockedForCheckPoint

  A user transaction has the status BlockedForCheckPoint at O when it is waiting for certain conditions to be satisfied to perform the checkpoint operation.

- Executing

  A user transaction is said to be executing at O if an operation $op(O)$ invoked by that transaction is executing

Figure 4.1 shows the Status Transition Graph which depicts the possible status transitions. When a request for an operation $op(O)$ is received at O for the first time from $Tr(op(O))$ and $op(O)$ starts execution, the status of $Tr(op(O))$ at O changes to Executing. When the operation completes all its steps, the status of $Tr(op(O))$ at O changes to Completed.

Some operations may have checkpoints specified for them. When an operation $op(O)$ reaches a checkpoint, if all the other active operations on O are at a checkpoint, then the status of $Tr(op(O))$ changes to CheckPointed. If all the active operations are not at their checkpoints, then $op(O)$ waits and the status of $Tr(op(O))$ changes to BlockedForCheckPoint. Thus, if an object has checkpoints specified for its operations, then the operations rendezvous at the checkpoints. If the checkpointing operation succeeds, there will be no future rollbacks for any operation beyond the checkpoint.

4.3.4 The protocol

The concurrency control protocol will be described in terms of the actions carried out at various points in time. This includes the protocol steps when the following occur:

- An operation is requested
- An operation calls another operation
- The called operation returns

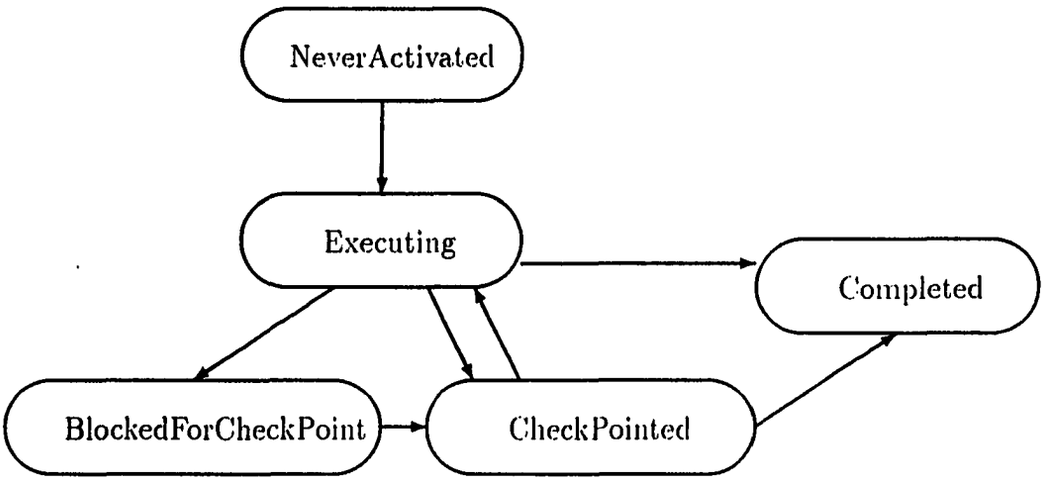

Figure 4.1: Status Transition Graph for a transaction at an object

- An operation is completed
- A breakpoint is reached
- A checkpoint is reached
- A checkpoint message is received

4.3.4.1 Protocol step – processing a request for an operation Consider an object O. Assume that operations $\{op_{j_1}, op_{j_2}, \ldots, op_{j_n}\}$ are currently executing at O. Let $CurPoint(op_j(O))$ denote the current step of an executing operation $op_j(O)$.

A request for an operation op on O is received (via a message) from an operation $op_i(O_i)$ (the calling operation). The message also contains the dependency information (in the form of a dependency set) from the calling object. The request for $op(O)$

is processed as follows:

- **Step 1.**

  Check if $DS(O) \cup DS(O_i)$ contains a cycle involving $Tr(op_i(O_i))$. If there is a cycle, then perform **Abort Processing** (described later). Otherwise, go to the next step.

- **Step 2.**

  Use the Conflict Table.

  (i) $\forall k, 1 \leq k \leq n$, if $E_{Con}(op(O), op_{j_k}(O)) = NO$.

  then schedule $op(O)$ for execution. No changes are necessary in any of the dependency lists. Also, status of $Tr(op(O))$ becomes Executing. If condition in 2(i) is not satisfied, go to step 2(ii); otherwise exit.

  (ii) $\forall k, 1 \leq k \leq n$, if $E_{Con}(op(O), op_{j_k}(O)) = NO$.

  or $CurPoint(op_{j_k}(O)) \in E_{BI}(op(O), op_{j_k}(O))$

  then:

  $\forall k, 1 \leq k \leq n$ such that

  $E_{Con}(op(O), op_{j_k}(O)) \neq NO$.

  add $\langle Tr(op(O)), Tr(op_{j_k}(O)) \rangle$ to $DS_L(O)$.

  If now $DG(O)$ contains a cycle involving $Tr(op(O))$, then perform **Abort Processing**. Otherwise, schedule $op(O)$ for execution and change status of $Tr(op(O))$ to Executing.

  (Note: We perform Abort Processing because "incorrectness" is imminent if we schedule $op(O)$ for execution.)

  If condition in 2(ii) is not satisfied, go to step 2(iii); otherwise exit.

(iii) $\forall k, 1 \leq k \leq n$, if $E_{Con}(op(O), op_{j_k}(O)) = NO$,
or $CurPoint(op_{j_k}(O)) \in E_{BI}(op(O), op_{j_k}(O))$
or $E_{Con}(op(O), op_{j_k}(O)) = Unknown$,
then:
$\forall k, 1 \leq k \leq n$ such that
$E_{Con}(op(O), op_{j_k}(O)) = YES$,
add $\langle Tr(op(O)), Tr(op_{j_k}(O)) \rangle$ to $DS_L(O)$.
If now DG(O) contains a cycle involving $Tr(op(O))$, then perform **Abort Processing**. Otherwise, schedule $op(O)$ for execution and change the status of $Tr(op(O))$ to Executing.
This execution will be validated later when the identities of the accessed objects are known and the dependency sets are propagated back with the results of the operations.
If condition in 2(iii) is not satisfied, go to step 2(iv); otherwise exit.

(iv) Block execution of $op(O)$ until each operation $op_{j_k}(O)$ for which $E_{Con}(op(O), op_{j_k}(O)) = Yes$, reaches a step such that $CurPoint(op_{j_k}(O)) \in E_{BI}(op(O), op_{j_k}(O))$. Then, go to step 2(ii).

4.3.4.2 Protocol step – calling an operation When an operation $op(O)$ executes a global step, i.e., sends a message to another object, the following steps are executed:

- The Dependency Set DS is constructed using the various dependency lists i.e., $DS(O) = DS_L(O) \cup DS_I(O) \cup DS_R(O)$.

- The message is sent to the receiving object, along with the message parameters and DS.

4.3.4.3 Protocol Step – called operation returns Let a global step of operation $op(O)$ result in a message to invoke operation $op_r(O_r)$. When the called operation $op_r(O_r)$ returns, a dependency list $DS(O_r)$ is also returned (along with the result of the method execution). The following steps are then performed:

- Add the dependency list $DS(O_r)$ to the list $DS_R(O)$.
- Construct the Dependency Graph and check for cycles involving the transaction $Tr(op(O))$. If such a cycle exists, perform **Abort Processing**.

4.3.4.4 Protocol Step – when an operation completes Let $op(O)$ be an operation invoked due to a message from an operation on O_i. The following steps are executed when $op(O)$ completes execution:

- Construct the Dependency Set $DS(O)$. Note that we can leave out the dependencies corresponding to $DS_I(O_i)$. DS is returned to O_i along with the result of operation $op(O)$.

4.3.4.5 Protocol Step – when a breakpoint is reached As noted earlier, the grouping and the breakpoint specification allows for certain interleavings between operations. This means that some of the dependency items from the local dependency list can be deleted when the operation reaches a breakpoint.

When an operation $op(O)$ reaches a breakpoint b_k, the following steps are performed:

- From the user-defined breakpoint specification, determine the level of the breakpoint, say l.

- From the grouping and breakpoint specifications, let $Op_b = \{op_{k_1}(O), op_{k_2}(O), \ldots, op_{k_m}(O)\}$ be the set of operations which can interleave op at b_k.

- Delete every dependency in $DS_L(O)$ which involves $Tr(op(O))$ and $Tr(op_{k_r}(O))$ where $op_{k_r}(O) \in Op_b$.

4.3.4.6 Protocol Step – when a checkpoint is reached The idea of checkpointing is used to limit the amount of rollback that is necessary when an operation has to be aborted. In our protocol, such operation aborts are sometimes necessary because of incorrect executions. The lack of a-priori knowledge about the objects which the operations access leads to incorrect operation interleavings. When a checkpoint is reached, it implies that future aborts of this or any other operation will not require a rollback beyond this point.

When an operation $op(O)$ reaches a checkpoint c_k, the following steps are performed:

- Construct $DS(O)$. Let $DepSet(Tr(op(O)), O) = \{T_j | \langle Tr(op(O)), T_j \rangle \in DS\}$ be the set of user transactions which conflict with and precede $op(O)$ (and hence $Tr(op(O))$) at O.

- $\forall j$ such that $T_j \in DepSet(Tr(op(O)), O)$, if either T_j has the status NeverActivated at O, or $op_k(O)$ (such that $Tr(op_k(O)) = T_j$) is at a checkpoint, go to

the next step. Otherwise, block until this condition is satisfied. The status of $Tr(op(O))$ is changed to BlockedForCheckPoint.

- A *checkpoint* message is sent to every object O_r which received a message from a global step of $op(O)$ to invoke $op_r(O_r)$. Change the status of $Tr(op(O))$ to CheckPointed.

4.3.4.7 Protocol Step – an object receives a checkpoint message When an object O receives a *checkpoint* message for $op(O)$ from an operation $op_i(O_i)$, the following steps are performed:

- Delete every dependency from $DS(O)$ which involves $Tr(op(O))$.
- Change the status of $Tr(op(O))$ to Completed.
- A *checkpoint* message is sent to every object O_r which received a message from a global step of $op(O)$ to invoke $op_r(O_r)$.

4.3.5 Abort Processing

Our locking protocol has an "optimistic" flavor to it. In the absence of a-priori knowledge about the bindings, the protocol takes the optimistic view that there may not be any incorrect behavior. The determination of incorrect behavior is done after the execution when all the bindings are known. If an incorrect execution of operations (and hence, user transactions) is noticed, we need to initiate corrective measures. We can rectify the incorrect execution by aborting the offending operation(s), undoing their effects and then re-executing them. Note that abort processing is initiated at a particular object.

4.3.5.1 Cycle detection and aborts Ideally, an abort should be processed as soon as an incorrect execution of operations is detected. In our protocol, the Dependency Set $DS(O)$ at an object O aids this process. The presence of cycles in the associated Dependency Graph $DG(O)$ implies the existence of an incorrect execution of operations. Clearly, an acyclic Dependency Graph can develop a cycle only when a new edge is added i.e., a new dependency is added to the Dependency Set. A new dependency may be added to DS in one of the following ways: adding a dependency to $DS_L(O)$, $DS_I(O)$ or $DS_R(O)$.

Consider two operations $op_i(O)$ and $op_j(O)$ with $Tr(op_i(O)) = T_i$ and $Tr(op_j(O)) = T_j$. Let us assume that the execution of a step of $op_i(O)$ (that conflicts and follows a step of $op_j(O)$) led to the addition of a dependency $\langle T_i, T_j \rangle$ to $DS(O)$ which resulted in a cycle in the dependency graph $DG(O)$. This implies that before the execution of this step, the Dependency Graph had a path from T_j to T_i. The addition of the new dependency resulted in the completion of a cycle. (The cycle may include other transactions in addition to T_i and T_j.) Since the existence of this cycle indicates an incorrect interleaved execution of the operations, it is necessary to abort at least one of the transactions and break the cycle. If the operation $op_j(O)$ is still executing at O, we can abort it and thus the dependency $\langle T_i, T_j \rangle$ will no longer be added. Aborting of $op_j(O)$ involves aborting the operations which may have been executed at other objects due to the global steps of $op_j(O)$.

However, if the operation $op_j(O)$ has already completed at O, we need to undo the results of $op_j(O)$ (in this, and all other objects which are accessed by the global steps of $op_j(O)$). Therefore, abort messages will have to be sent to all these objects. The receiving objects will in turn need to rollback the effects of the aborted oper-

ations and all other operations dependent on them. Note that we do not intend to describe how the information needed for rollbacks is obtained. There exists extensive literature [34, 35, 36] on maintaining logs (undo/redo logs) which can aid the process of rollbacks.

4.3.5.2 Handling abort requests When an object O receives an abort request for a particular operation, say $op_j(O)$, the following steps are performed:

- If $op_j(O)$ is currently executing at O.
  - Abort the operation $op_j(O)$
  - Undo the local effects of the steps of $op_j(O)$
  - Send an abort message to all objects which received a message from O as a result of a global step of $op_j(O)$.
- If $op_j(O)$ has finished execution at O.
  - Undo the local effects of the steps of $op_j(O)$
  - Send an abort message to all objects which received a message from O as a result of a global step of $op_j(O)$.
  - Send an abort request to the calling object i.e., the object that contained the operation whose global step invoked $op_j(O)$.
- For each operation $op_i(O)$ which conflicts with and follows $op_j(O)$ at O:
  - Send an abort request to O (self) to abort $op_i(O)$

Abort messages can thus be used to achieve a rollback that is needed when an incorrect execution of operations is observed.

Note that aborting an operation can have a cascading effect because of the dependence of operations on the results of other operations. The discussion of means to limit the cascading effect is beyond the scope of this dissertation.

4.4 An Example

To illustrate how the locking protocol works, let us consider an example of a MMDB system. Albeit contrived and oversimplified, the example demonstrates the operation of the protocol. One of the components of a multimedia system is a movie object. Each movie object consists of an audio component, a video component and some synchronization information. The users may perform creating, editing and previewing operations with the aid of movie editors and previewers. The user transactions interact with the MultiMedia object. The following is a partial description of some of the classes in the system and their associated parts (instance variables) and methods (operations). The syntax used is similar to that in ABC, an object-oriented system developed at Iowa State University. In the sample definitions, only the relevant parts or methods are shown.

```
CLASS MultiMedia
   PARTS
   METHODS
     EditMovie moviename:String with movie_editor:MovieEditor
       {
         :
         movie_editor EditMovie moviename.
         :
       }.
     PreviewMovie moviename:String with previewer:Previewer
       {
         :
         previewer PreviewMovie moviename.
         :
       }.
     CreateMovie moviename:String with movie_editor:MovieEditor
       {
         :
         movie_editor CreateMovie moviename.
         :
       }.
     ListEquipment
       {
         :
       }.
ENDCLASS.
```

```
CLASS MovieEditor
   PARTS
   METHODS
     CreateMovie moviename:String
       {
         :
         movie := directory1 GetNamedObject moviename.
         (movie's audio) EditAudio.                    (C1)
         (movie's video) EditVideo.                    (C2)
         (movie's sync_info) EditSyncInfo.             (C3)
       }.
     EditMovie moviename:String
       {
         movie := directory1 GetNamedObject moviename.
         (movie's audio) EditAudio.                    (E1)
         (movie's video) EditVideo.                    (E2)
         (movie's sync_info) EditSyncInfo.             (E3)
       }.
ENDCLASS.
```

```
CLASS Previewer
   PARTS
   METHODS
     PreviewMovie moviename:String
       {
         movie := directory1 GetNamedObject moviename.
         (movie's video) ReviewVideo.                  (P1)
         (movie's audio) ReviewAudio.                  (P2)
         (movie's sync_info) ReviewSyncInfo.           (P3)
       }.
ENDCLASS.
```

```
CLASS Directory
   PARTS
   METHODS
     GetNamedObject name:String
       {
         :
         RETURN fetched_object.
       }.
ENDCLASS.
```

```
CLASS Movie
   PARTS
     audioclip:Audio
     videoclip:Video
     sync_info:Synchronization
   METHODS
ENDCLASS.
```

```
CLASS Audio
   PARTS
   METHODS
     EditAudio
       {
         :
       }.
     ReviewAudio
       {
         :
       }.
ENDCLASS.
```

```
CLASS Video
   PARTS
   METHODS
     EditVideo
       {
         ⋮
       }.
     ReviewVideo
       {
         ⋮
       }.
ENDCLASS.
```

```
CLASS Synchronization
   PARTS
   METHODS
     EditSyncInfo
       {
         ⋮
       }.
     ReviewSyncInfo
       {
         ⋮
       }.
ENDCLASS.
```

Assume that there are instances MMObject (*Class: MultiMedia*), directory1 (*Class: Directory*), Editor1 (*Class: MovieEditor*), Previewer1 (*Class: Previewer*), Movie1 (*Class: Movie*), Audio1 (*Class: Audio*), Video1 (*Class: Video*), and Sync1 (*Class: Synchronization*).

Assume the following:

- For MultiMedia, $E_{Con}(PreviewMovie, EditMovie) = Unknown$.
  Since the movie object is dynamically determined, it is not known a-priori if PreviewMovie and EditMovie conflict.

- For MultiMedia, $E_{Con}(ListEquipment, EditMovie) = No$.
  ListEquipment simply provides a list of the available multimedia equipment and hence does not conflict with other operations.

- MovieEditor has a 3-level grouping and breakpoint specification such that the steps in CreateMovie can interleave those in EditMovie at E_1, E_2 and E_3, while the steps in EditMovie can interleave those in CreateMovie at C_1, C_2 and C_3. Also, for MovieEditor, $E_{Con}(CreateMovie, EditMovie) = Yes$.

- EditAudio and ReviewAudio operations conflict with each other, i.e., for Audio $E_{Con}(EditAudio, ReviewAudio) = E_{Con}(ReviewAudio, EditAudio) = Yes$.

- EditVideo and ReviewVideo operations conflict with each other.

- EditSyncInfo and ReviewSyncInfo operations conflict with each other.

Note that the operations in MovieEditor first access the audio object, followed by the video object and then the synchronization information. However, the Previewer first accesses the video object, followed by the audio object and the synchronization information.

Consider the user transactions T_{edit}, T_{prev}, T_{create} and T_{list} that invoke the operations $EditMovie(MMObject)$, $PreviewMovie(MMObject)$, $CreateMovie(MMObject)$ and $ListEquipment(MMObject)$ respectively.

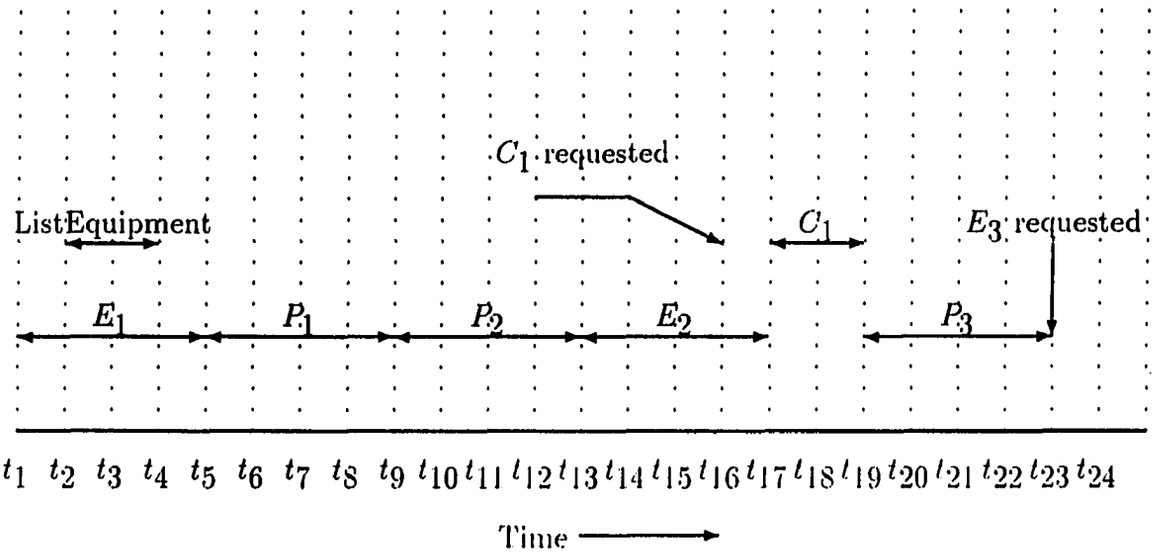

Figure 4.2: An example schedule

Figure 4.2 shows an example of a schedule. Let us examine how the protocol works. Assume that the instance of MovieEditor used by T_{edit} and T_{create} is Editor1; Previewer1 is the instance of Previewer used by T_{Prev}. Assume that the instance of movie used is Movie1, which contains Audio1, Video1 and Sync1 as the instance variables. Note that the identity of the instance of movie is not known a-priori. Figure 4.3 partially illustrates the message passing which occurs as a result of the operations on MMObject.

To begin with, all the dependency sets are empty. Transaction T_{edit} invokes the EditMovie(MMObject) operation which results in the invocation of EditMovie(Editor1) operation (shown as E_1 in the figure). While E_1 is executing, T_{list} requests the ListEquipment(MMObject) operation. Since $E_{Com}(ListEquipment, EditMovie) = No$, ListEquipment(MMObject) can be executed concurrently (Step 2(i) of the protocol).

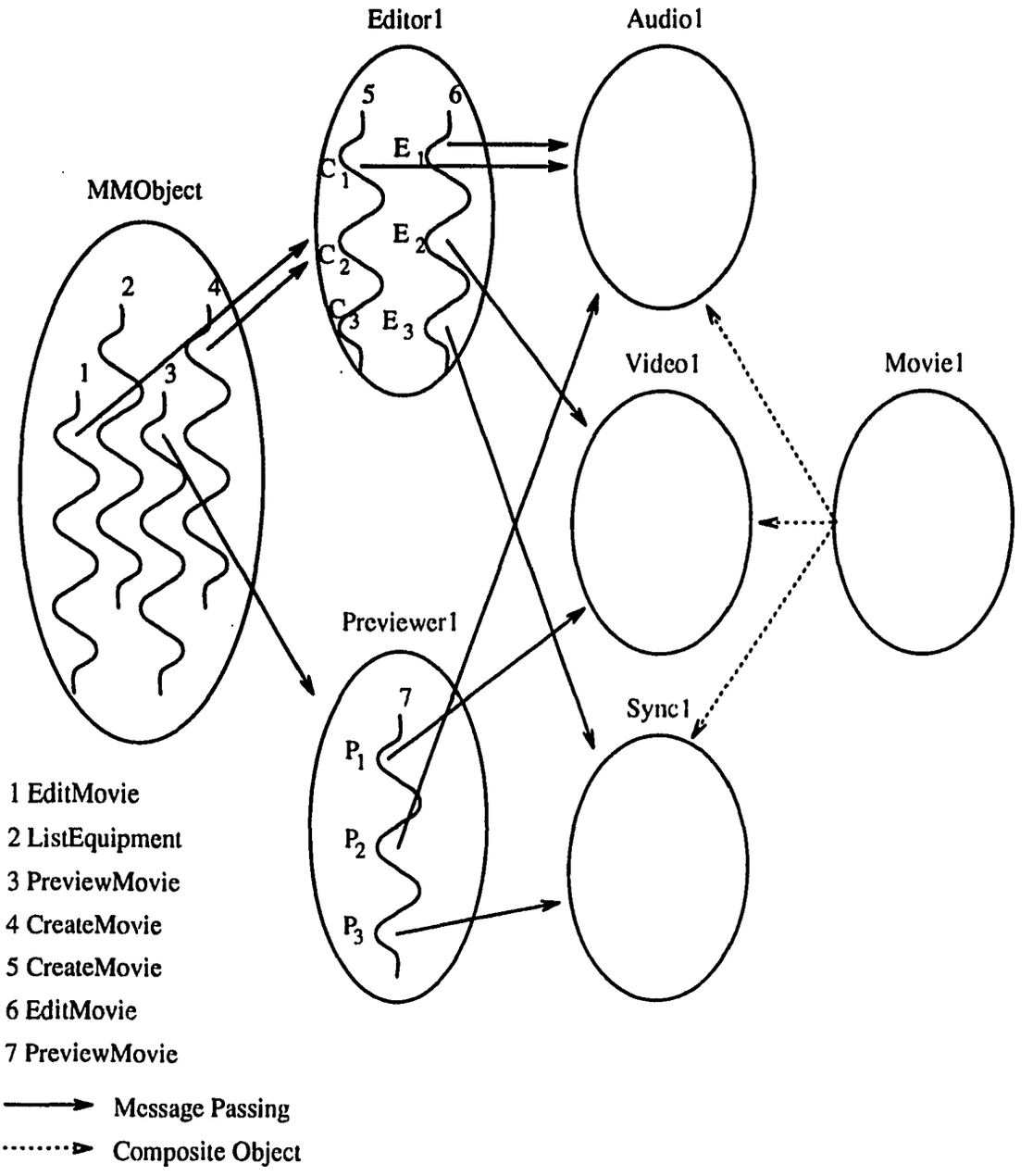

Figure 4.3: Example method invocations

At t_5, T_{prev} requests PreviewMovie(MMObject). Since $E_{Con}(PreviewMovie, EditMovie) = Unknown$, Step 2(iii) allows the scheduling of PreviewMovie(MMObject) which in turn invokes PreviewMovie(Previewer1). Thus P_1 is scheduled. At t_9, P_2 is scheduled. Since T_{edit} is still not completed, the dependency $T_{prev} \rightarrow T_{edit}$ is added to $DS_L(Audio1)$. On completion of P_2, this dependency is returned to Previewer1. When E_2 accesses Video1, the dependency $T_{edit} \rightarrow T_{prev}$ is added at Video1 and then returned to Editor1. Note that T_{edit} and T_{prev} have accessed the objects Audio1 and Video1 in conflicting order.

While E_2 is still executing, T_{create} is initiated and results in a request for CreateMovie(Editor1) at t_{16}. However, the protocol requires C_1 to block (Step 2(iv)). At t_{17}, a breakpoint of EditMovie is reached and Step 2(ii) allows the scheduling of C_1. P_3 then accesses Sync1. The dependency $T_{prev} \rightarrow T_{edit}$ is received at Sync1 from Previewer1. After completion of P_3, when E_3 accesses Sync1, the dependency $T_{edit} \rightarrow T_{prev}$ is also received at Sync1. Sync1 now notices that T_{edit} and T_{prev} have incorrectly accessed the Movie object (Step 1) and hence initiates abort processing. The abort processing is necessary since T_{edit} accessed Video1 after T_{prev} and hence the synchronization information in Sync1 may no longer be correct.

4.5 Proof of Correctness

An object-oriented correct schedule (OOCS) is defined in the previous chapter as one that satisfies the conditions imposed by the grouping and breakpoint specifications. In other words, an OOCS can have only those interleavings which are allowed by the breakpoint and grouping specifications. An object-oriented correctable schedule (OOCLS) is an object-oriented schedule which is equivalent to some OOCS.

Theorem 1

The proposed concurrency control protocol allows only object-oriented correctable schedules (OOCLS).

Proof

To prove Theorem 1, let us consider the steps followed by the locking protocol. We will show that the only interleavings allowed are those that can be found in an OOCLS. We use the notations introduced in the previous chapter.

Let Sch be an object-oriented schedule. Consider an object O and operations $op(O)$ and $op'(O)$ that occur in Sch. Assume that user transaction T requests the execution of $op(O)$. While $op(O)$ is executing, user transaction T' requests the execution of $op'(O)$. The grouping and breakpoint specifications allow $op'(O)$ to interleave only when $op(O)$ is at a breakpoint visible to $op'(O)$.

When the request for $op'(O)$ is received, we have the following two cases.

- $op(O)$ is at a breakpoint visible to $op'(O)$:
  In this case, $op'(O)$ can be scheduled for execution. Either Step 2(i) or 2(ii) will allow $op'(O)$ to be scheduled for execution.

- $op(O)$ is at a step other than a breakpoint visible to $op'(O)$:
  We have three subcases:

  - $E_{Con}(op'(O), op(O)) = NO$
    i.e., $op(O)$ and $op'(O)$ do not conflict. In this case, we can still schedule $op'(O)$ for execution (Step 2(i)). If the two operations do not conflict (i.e., none of their steps conflict), then they can be executed in any order without affecting the final result. (See the previous chapter for further

details.)

- $E_{Con}(op'(O), op(O)) = Yes$

  In this case, $op'(O)$ is blocked (Step 2(iv)) and thus an incorrect interleaving is not allowed.

- $E_{Con}(op'(O), op(O)) = Unknown$

  This is the case where the conflict information is not available a-priori and the protocol takes an optimistic approach. It needs to be shown that even if $op'(O)$ is scheduled for execution, an incorrect execution will not occur. Let $Level(op(O), op'(O)) = l$; $s_i, s_j \in b_l; b_l \in B_{op(O)}(l); s_k \in b'_l; b'_l \in B_{op'(O)}(l); s_i <_{op(O)} s_j$. We have the following possibilities.

  * s_k does not conflict with either of s_i or s_j.

    In this case, any order of execution of these steps gives the same final result.

  * s_k conflicts with s_i but not with s_j.

    In this case, an OOCS which is equivalent to Sch will have both of s_i and s_j either preceding or following s_k depending on whether s_i precedes or follows s_k in Sch.

  * s_k conflicts with s_j but not with s_i.

    Same as the previous case.

  * s_k conflicts with s_i and s_j.

    If s_k follows or precedes both of s_i and s_j in Sch, an equivalent OOCS will also have s_k following or preceding them. However, if $s_i <_{Sch} s_k <_{Sch} s_j$, then we cannot find an equivalent OOCS (by definition

of an OOCS). However, our protocol does not allow such an execution. Suppose that $op(O)$ has executed step s_i when the request for $op'(O)$ is received. Step 2(iii) of the protocol will allow $op'(O)$ to be scheduled for execution. At some point, $op'(O)$ executes step s_k. This will result in the dependency $\langle Tr(op'(O)), Tr(op(O))\rangle$ to be returned to O. So we have $s_i <_{Sch} s_k$. Now assume that s_j is executed. Since s_j conflicts with and follows s_k, the dependency $\langle Tr(op(O)), Tr(op'(O))\rangle$ will be returned to O. When the called operation (or the global step s_j) returns, the dependency graph is checked for cycles. Since such a cycle exists, abort processing will be initiated. Thus, the protocol does not allow an incorrect interleaving such as $s_i <_{Sch} s_k <_{Sch} s_j$.

Thus we have shown that the locking protocol indeed allows only correctable schedules. □

4.6 Comparison with Existing Protocols

As described in section 4.2.2, the existing literature on concurrency control fails to meet the requirements of the advanced applications that use object-oriented data base systems. In this section, we briefly compare our protocol against the existing ones in the current literature. The comparison shown in Table 4.4 uses the same criteria as Table 4.1. For ease in reading, we restate the criteria.

- Handle long-duration cooperating transactions (LO)
- Enforce correctness criterion other than serializability (CO)
- Deal with object-oriented features (OO)

Table 4.4: Comparison of our protocol against existing literature

	LO	CO	OO	GO	LB
[5]	No	No	Yes	No	No
[16]	No	No	Yes	No	No
[31]	Yes	Yes	No	No	No
[32]	Yes	Yes	No	No	No
[30]	Yes	Yes	No	No	No
New	Yes	Yes	Yes	Yes	Yes

- Allow generalized operations (i.e., other than read and write) (GO)
- Handle late binding (LB)

Note that we had not considered LB, the issue of late binding, in our earlier comparison because none of the discussed literature addresses this issue.

As can be seen from the table, the concurrency control mechanisms suggested in this chapter meet all the requirements stated above. It should be noted that although [16] supports the object-oriented data model, it does not handle all the object-oriented features such as the class and the class-composition hierarchies. Our transaction model described in the previous chapter and the concurrency control protocol presented here, provide support for object-oriented features and also handle long-duration cooperating transactions that occur in advanced applications. The semantic richness of the object-oriented model is exploited via the means of grouping and breakpoint specifications described earlier.

4.7 Summary

This chapter presented a concurrency control protocol for object-oriented data base systems. The advanced applications which use these data bases are characterized by long-duration cooperating transactions. The existing concurrency control protocols in the current literature do not address the unique requirements of these applications.

We would like to emphasize the need for further research in the area of transaction management in object-oriented data base systems. In particular, further research is needed in the areas of deadlock detection, avoiding cascaded aborts and implementation issues. The coming years will see a manifold increase in the use of object-oriented data base systems, especially with the advent of advanced applications which require the semantic richness provided by the object-oriented model. The users of these data bases will interact with them through various query languages. In the next chapter, we identify some of the issues which such query languages for object-oriented data bases will need to address and suggest some solutions.

5. QUERY PROCESSING

5.1 Introduction

Data base systems allow users to store large amounts of information. Query languages are an important tool in aiding users to retrieve this information in a meaningful way. Query languages provide a convenient means for average users to interact with the data base. In relational systems, declarative query languages are popular because of their simplicity and ease of use. There is considerable ongoing research in the area of query languages for object-oriented data bases [37, 18, 38, 39, 40].

The user queries have to be translated into operations on the underlying data base. These operations (grouped into transactions) interact with the data base system and return the results back to the user. The queries may either be translated through an interpreter or compiled through a compiler. A query interpreter allows for interactive queries while a compiler allows for a more efficient data base access in batch mode. Relational systems usually do not have an associated programming language. In such cases, the query language provides a means for access to the data. In object-oriented data bases, an object-oriented programming language usually is an integral aspect of the system. Thus, the programming language can itself act as a means to access the data. The query language then provides a convenient mechanism

for user to interact with the underlying programming language.

5.1.1 Our goals

The design of an object-oriented query language is an important aspect of object-oriented data bases. In this chapter, we survey some of the existing query languages for object-oriented data bases and identify some of the important issues most query languages are expected to face in the object-oriented context. Some of these issues include:

- Query Translation
- Retrieval of objects, creation of new objects, modification and deletion of existing objects
- Modification of class and class-composition hierarchies

Object-oriented data base systems will need to provide the support necessary for query languages. Such support includes the facility to create and store new objects, and to modify existing objects. In particular, the data base will have to support the modification of the class and the class-composition hierarchies.

In section 5.2, we describe the current research in the area of query languages for object-oriented data base systems. Section 5.3 describes the object-oriented query model. The translation of a query to operations on the underlying data base is described in section 5.4. Queries on a data base may lead to the creation, deletion and modification of objects in the data base. These issues are discussed in section 5.5. Object-oriented data bases provide the users with the capability to dynamically reorganize the structure of data. Section 5.6 discusses modification of the class and

the class-composition hierarchies in systems with single inheritance and section 5.7 discusses the same issues for systems with multiple inheritance. Section 5.8 provides a few concluding remarks.

5.2 Background

In an object-oriented system, access and retrieval of data is navigational in nature. This is because of the existence of complex objects and their relationships with other objects. Since the data is encapsulated, direct access to data is not permitted. Messages are the only means to access the data. A message to an object may in turn result in messages to other objects. This is in contrast with relational systems with direct access to data. Query languages are used to facilitate the access to data by attempting to hide the navigational nature.

A representative selection of literature on query languages for object-oriented data bases includes [37, 18, 38, 39, 40]. In this section we will briefly summarize their work.

In [37], the concept of views as applicable to object-oriented data bases is discussed. A view is just a query on the data base. The view mechanism is based on the O_2 model [41]. The authors present a mechanism which allows a programmer to restructure the class hierarchy and modify the behavior and structure of objects. The concept of virtual classes is discussed with examples.

Straube and Ozsu define an object-oriented data model and a query model [40]. The authors present an object calculus and a closed object algebra. Issues of query safety and translation of object calculus to object algebra are discussed. Equivalence-preserving transformation rules for object algebra expressions are also provided.

The problem of designing a query language that provides an ad hoc querying facility and integrating it in the overall object-oriented data base system is discussed in [18]. An ad hoc querying facility, in the form of a query language, is needed to enable users to extract information from the data base without having to write programs in the object-oriented programming language.

In [38], Beech defines Object SQL (OSQL), which reinterprets and extends SQL to define object types and instances. Furthermore, functions which relate and manipulate objects are also discussed. One of the goals in defining OSQL is to ease the migration from relational to object data bases. However, the full power of object-oriented data bases is not exploited.

In [39], the authors discuss CQL++, which is a SQL for the Ode object-oriented DBMS. A SQL-like syntax and the C++ class model are combined to provide a declarative front end to Ode. CQL++ provides facilities for manipulating classes and objects. The data base user is provided a relatively straightforward interface by hiding details of object-oriented features of O++, the underlying object-oriented programming language with persistence. Like most other extensions of SQL to object-oriented data bases, CQL++ is also designed to facilitate an easy migration for traditional users of relational systems to object-oriented data bases.

Kim presents a query model and a query language for the ORION object-oriented DBMS in [5]. The concept of schema graphs and query graphs are discussed. The query model has been designed to reflect the semantics of the class hierarchy. Object-oriented equivalents of the relational operations such as projection, selection, join and set are discussed.

In this chapter, we will be examining some of the issues which will be encountered

by the query languages in an object-oriented environment. The complex object-oriented data model with its semantic richness poses a different set of challenges than those in the relational world. In relational systems, some of the important issues included those of forming efficient joins and maintaining normal forms. In the object-oriented domain, the class and class-composition hierarchies have an important impact on the retrieval, modification and creation of data (objects).

5.3 Query Model

The object-oriented data base architecture differs from the relational architecture. Object-oriented features such as encapsulation, class and class-composition hierarchies, and complex object structure differentiate the nature of the stored data. In relational systems, the data is primarily stored in terms of relations or tables with a very regular structure. The task of interpretation of such data is easier compared to that of the complex data in an object-oriented system.

Object-oriented programming languages are well equipped to manage the objects and interpret their complex structure correctly. An object-oriented programming language enhanced with the ability to handle persistent data (the object repository) provides a suitable means for managing an object-oriented data base system (Figure 5.1). However, being a full-fledged programming language, such a language may be too complex for ordinary data base users. Therefore, some form of a query language is needed to provide users with an alternative means to access the data base.

Declarative languages such as SQL are very popular in relational data bases. As noted earlier, initial attempts in the area of query languages for object-oriented data bases have focussed on the extension of relational query languages to object-

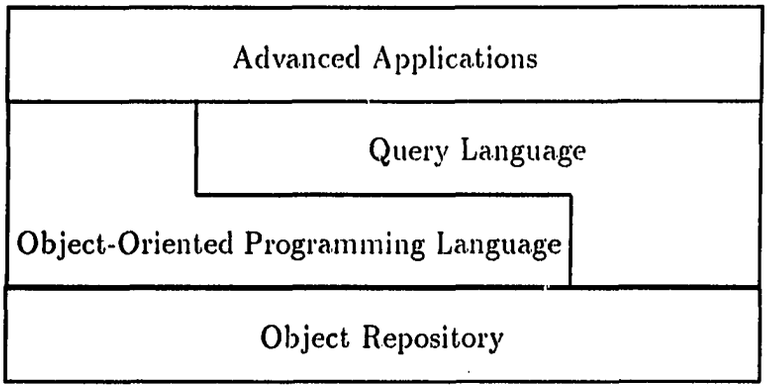

Figure 5.1: Object-oriented Data Base Architecture

oriented data bases. However, such an extension is restrictive in nature. While on one hand it does facilitate the transition of traditional relational data base users to the object-oriented realm, it also fails to exploit the full power of the object-oriented features.

In general, we expect the object-oriented query language to be a simplification of the full power of the object-oriented programming language. Ordinary data base users do not need the full computational power provided by the object-oriented programming language. However, some other features of the programming language may be incorporated into the query language. For example, the query language may understand the complex structure of the objects. Thus, the query language could be a subset of the programming language. The query language may also simplify some of the complexities of the programming language by sacrificing some of its power. It can thus be built on top of the programming language. Figure 5.1 illustrates this mixed nature of the query language: partially built on top of the programming language, and at the same time retaining some direct access to the objects similar to

the programming language.

It may be pertinent to mention the issue of embedding query languages in some other language. This issue is important in relational systems where query languages have limited computational power. Tasks which require intense computational processing of the stored data are usually performed using languages such as C enhanced with embedded query language constructs. The embedded query language constructs facilitate the retrieval of stored data which is then used in computations. In the case of object-oriented data bases, such embedding of query languages is not important. This is because the object-oriented programming language is itself powerful enough to perform the required processing [18].

5.4 Query Translation

As can be seen from Figure 5.1, query language operations have to be mapped to the underlying object-oriented programming language. This is necessary to correctly interpret the complex structure of the objects. The objects can be manipulated through messages which define their interface.

A typical SQL-like query has the following form:

SELECT S **FROM** C **WHERE** P.

The meanings of the symbols are:

- S is some collection of objects. This collection can have any arbitrary format and the objects may not necessarily be instances of any of the existing classes in the data base. The objects may be an arbitrary partial collection of instance variables of existing objects, or, may be complete existing objects.

- C is a list of one or more classes which forms the search space for the predicate P. These classes may either exist in the data base or may be the result of some other query

- P is a predicate which denotes the condition(s) that are necessary for objects to be selected.

The above SQL-like query will be translated into messages which are sent to appropriate objects. These objects then execute the methods associated with the messages and return the results. For example, if the query requests the salary information of employees who earn more than \$70,000, messages will be sent to the instances of employee objects (or maybe to the class Employee) to retrieve the salary information. The salaries are checked to see if they exceed \$70,000. All the employee objects meeting this criterion are then returned as the result of the user query.

In cases where objects in S do not belong to an existing class, new classes may need to be created. Such classes may be either temporary (exist only for the duration of the query session) or permanent. This issue is further discussed in the next section.

In general, user queries may not be restricted to mere retrieval of information. Queries may also modify existing objects, create new objects, and delete existing objects. In particular, users of object-oriented data base systems may modify existing classes, create new classes, and delete existing classes. Such operations on a class hierarchy also affect the objects that are instances of these classes.

In chapter 3, we had classified the effects of any operation on an object into four major categories:

- Query

  This is the more traditional notion of queries wherein their main function is the retrieval of information from the data base

- Create

- Modify

- Delete

The user queries are translated into operations on objects. The operations thus result in retrieval, creation, modification or deletion of objects. To support user queries, the data base needs to provide support for these operations. Sections 5.5, 5.6 and 5.7 describe the data base support needed to support user queries. For ease of understanding, class objects are discussed separately from instance objects. Section 5.5 discusses the data base support needed for manipulating instance objects. Sections 5.6 and 5.7 discuss the support needed for manipulating class objects.

5.5 Retrieval, Creation, Modification and Deletion of Objects

In addition to retrieving stored information from the data base, queries may also be used to modify the information or add new information to the data base. Thus, a query on a data base may result in one or more of the following actions:

- Retrieval of existing objects

- Creation of new objects

- Modification of existing objects

- Deletion of existing objects

5.5.1 Retrieval of objects

A query may result in retrieval of some collection of objects from the data base. In an object-oriented system, every object is an instance of some class. Therefore, each object returned as a result of the query must also be an instance of some class. Consider the class Movie shown below.

```
CLASS Movie
   PARTS
     moviename:String
     audioclip:Audio
     videoclip:Video
     creator:Person
     creation_date:Date
     sync_info:Synchronization
ENDCLASS.
```

Suppose a query requests only the names and creators of movies meeting a certain criterion, then the objects returned to the query may not belong to any existing class in the data base. This will require the creation of a class, with only the moviename and creator as the attributes (parts).

```
CLASS TempClass
   PARTS
     moviename:String
     creator:Person
ENDCLASS.
```

TempClass needs to exist only for the duration of the query session. This class will be a top-level class and will have no methods of its own. The objects returned to

the query are thus considered to be instances of TempClass. Note that this class will be visible only to the data base transaction representing this query. Thus, if there are two different non-cooperating transactions of the above kind, each will lead to a creation of a separate temporary class.

5.5.2 Creation of objects

The user queries may create new objects that are instances of an existing class. In some cases, the creation of such objects is implicit. For example, when the result of a query contains objects which do not belong to an existing class, a temporary class is created and the instance objects are implicitly created. Such objects are temporary in nature and exist only for the duration of the query session. Once the session is completed, the objects are removed from the data base. Also, during their lifetime, such objects are not visible to any transaction other than the one representing the query which led to their creation.

The users of an object-oriented data base may also explicitly create new objects of an existing class. When creating instances of an existing class, the underlying object-oriented system needs to generate appropriate object identifiers for the new objects. These objects are then initialized with user supplied values (or with appropriate default values when the user does not supply them). These objects are permanent in the sense that they continue to exist in the data base even after the query session is completed. These objects are also visible to transactions which follow the transaction that created them. Thus they are a shared entity and subject to the usual concurrency control mechanisms.

5.5.3 Modification of objects

When a user query modifies an existing instance object in the data base, no new classes or objects have to be created. The data base transaction representing the query is subject to the usual concurrency control requirements, i.e., the transaction execution will have to obey the concurrency control mechanisms. The modification of class objects is a more complicated issue and will be considered separately in sections 5.6 and 5.7.

5.5.4 Deletion of objects

Deletion of an object by a user query will result in its removal from the data base. This deletion is permanent and the object is not available to any of the transactions which follow the transaction that deleted it. Note that the deletion of an object does not imply the deletion of all the other objects that are referred to by the deleted object. The impact of deletion of a class object is more complex and is discussed in the following section.

5.6 Retrieval, Creation, Modification and Deletion of Classes – Single Inheritance

The retrieval of class objects is not much different from that of instance objects. The user queries may request information about a class or the class hierarchy rooted at a particular class. In most object-oriented systems, such information is readily available. The focus of this section is on the creation, modification and deletion of classes. These operations not only affect the classes involved, but also the class and the class-composition hierarchies in which the involved classes participate.

One of the features of an object-oriented data base is the ability to modify the structure of the data base. In a relational system, modification of the data base schema is not a trivial operation. On the other hand, modification of classes (the object-oriented analog of relations) is supported by most object-oriented systems.

Users of object-oriented data base may request the reorganization of the information in the data base by means of modifying, adding or deleting classes from the class hierarchy. We expect most query languages for object-oriented data bases to provide this functionality in one form or another. Users of traditional relational system may not need to use this feature. However, advanced applications that are expected to use object-oriented data bases (such as CAD and MMDB systems) will likely require the ability to dynamically reorganize the structure of the data base i.e., make changes to the class and class-composition hierarchies.

Before we describe the impact of creation, modification and deletion of classes, we introduce the concept of conflict resolution. Object-oriented systems are characterized by inheritance. Subclasses inherit the components (instance variables) and the methods defined for their parent classes. We will restrict our initial discussion to single inheritance; multiple inheritance will be discussed later. In addition to inheriting the characteristics of the parent classes, new instance variables and methods may be defined for a class or the existing ones may be modified. In such cases, we propose the following conflict resolution rule for instance variables and methods.

Conflict resolution rule for instance variables and methods:

- If the new instance variable or method has the same name as an existing instance variable or method inherited from the parent class, the new definition of the instance variable or method takes precedence over the inherited definition.

5.6.1 Creation of classes

In section 5.5.1, we discussed the implicit creation of temporary classes as a result of a user query. The user query may also explicitly request the creation of a new class. In such cases, it is expected that the user will provide all the necessary details regarding the position of the new class in the class hierarchy as well as new methods and instance variables for this class. This new class will then be visible to all the transactions following the one which created it. For example, let us revisit the class Movie defined below.

```
CLASS Movie
   PARTS
     moviename:String
     audioclip:Audio
     videoclip:Video
     creator:Person
     creation_date:Date
     sync_info:Synchronization
ENDCLASS.
```

The user may now request the creation of a new class called DubbedMovie that has the class Movie as its parent (Figure 5.2).

```
CLASS DubbedMovie
   PARENT Movie
   PARTS
     dubbedtext:VideoText
ENDCLASS.
```

This enables the class DubbedMovie to inherit the behavioral aspects of class Movie and add some of its own. For example, DubbedMovie inherits the instance

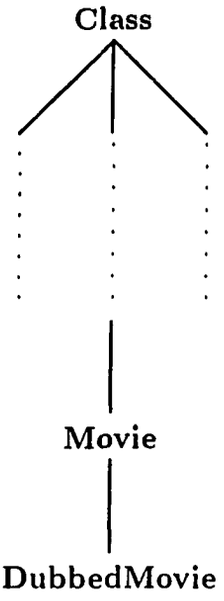

Figure 5.2: Creating a leaf class - partial class hierarchy

variables moviename, audioclip, videoclip, creator, creation_date and sync_info which are defined for the class Movie. Note that this newly created class will be added as a leaf in the class hierarchy and the conflict resolution mechanisms will be applied. The insertion of a class at a non-leaf node in the class hierarchy is a more complicated operation. Suppose that class C_j is a subclass of C_i and a new class C_k is inserted in between, i.e., C_k becomes the parent of C_j and C_i becomes the parent of C_k (Figure 5.3). In this case, the conflict resolution rules have to be applied to all the classes in the class hierarchy rooted at C_k. This implies that all the existing instances of these classes may also have to be modified to reflect these changes. For example, if a new instance variable is defined in C_k, all the instances of all classes in the class hierarchy rooted at C_k will acquire the new instance variable, initialized to the

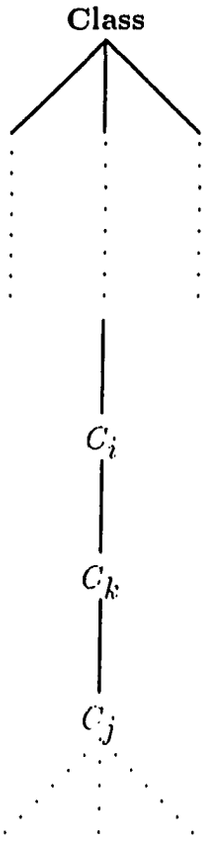

Figure 5.3: Creating a non-leaf class - partial class hierarchy

default value.

During the creation of the new class, the concurrency control mechanisms will restrict access to the objects in the system which are likely to be affected, especially the class and class-composition hierarchies. In general, operations that affect the class and class-composition hierarchies are expected to be infrequent. Since these operations impact considerable number of objects in the system, the transactions representing them are not expected to cooperate with any other transactions in the

system. In terms of concurrency control, such transactions will need to be executed in isolation and will require the blocking of other transactions in the data base. Since any modification to the class and class-composition hierarchies potentially affects the instances of the modified classes (and their subclasses), it is recommended that such transactions be carried out in the absence of any conflicting transactions. Such an isolated execution is not expected to degrade the overall performance of the system because of the relatively low frequency of such transactions.

5.6.2 Modification of classes

The modification of a class C not only affects C but also the entire class hierarchy rooted at C and the instances of all such classes. In data base terms, this is a major reorganization of the information structure and therefore such a transaction is expected to be executed with no other concurrent conflicting operations.

For example, let us consider the class Movie again. Suppose the user wishes to modify the class definition to be as follows:

```
CLASS Movie
   PARTS
     moviename:String
     audioclip:Audio
     videoclip:Video
     creator:Person
     creation_date:Date
     sync_info:Synchronization
     revision_history:Text
ENDCLASS.
```

This change to class Movie not only affects all the instances of this class but also the instances of class DubbedMovie (and all other classes in the class hierarchy rooted at class Movie). All these instance objects will require the addition of an additional part. Typically, this involves the deletion of the existing instance and creation of a new instance with the correct number of parts. Thus, all the objects in the system which have references to the deleted instances will need to be furnished with the new object identifiers. In general, the modification of a class will require the application of the conflict resolution rules to the classes in the class hierarchy rooted at the modified class and also a modification of the instances of the affected classes.

In addition to affecting the class hierarchy, the modification of a class also affects the class-composition hierarchy. For example, consider the following definition for class Audio.

```
CLASS Audio
   PARTS
     audiofilename:String
     cliplength:Integer
ENDCLASS.
```

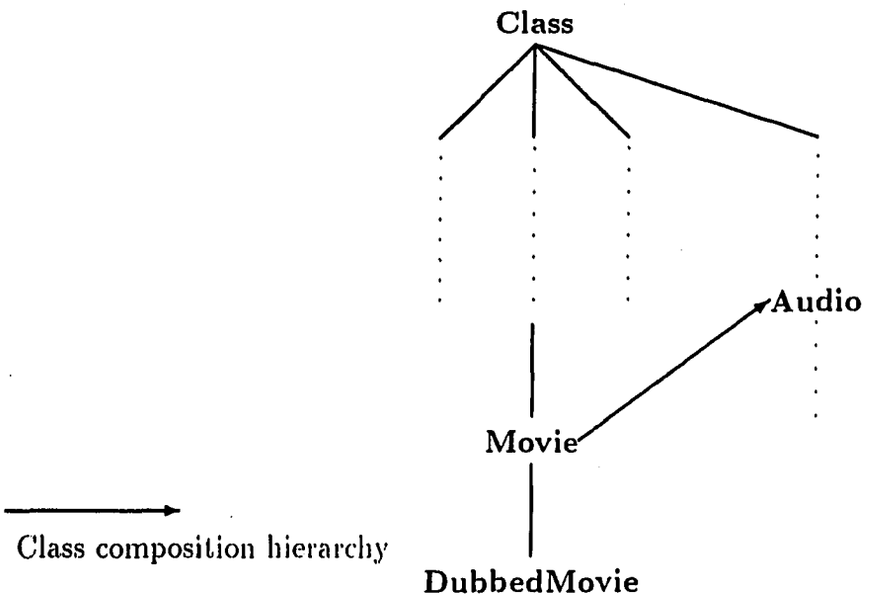

Figure 5.4: Modification of a class - partial class and class-composition hierarchies

Note that the class Movie has an instance variable of class Audio. This relationship will appear in the class-composition hierarchy. Now, suppose the definition of class Audio is modified to the one shown below.

```
CLASS Audio
   PARTS
     audiofilename:String
     cliplength:Integer
     editing_history:Text
ENDCLASS.
```

In this case, this modification not only affects the class hierarchy rooted at class Audio, but also affects classes such as Movie because of the class-composition hierar-

chy (Figure 5.4). Since the instances of class Audio have to be modified and possibly assigned new object identifiers, the references to these instances in instances of class Movie also need to be updated. In addition to affecting the instance objects via the class and class-composition hierarchies, the modification of a class may require the modification of the methods defined for the affected classes. For example, methods referring to attributes that are dropped from a class will have to be modified. This modification of methods is not expected to be an automated process because it requires the user to specify the required changes. However, the object-oriented system should notify the user regarding the methods that need to be modified.

5.6.3 Deletion of classes

Like modification, deletion of a class C also affects the class hierarchy rooted at C. All the subclasses of C have to be reorganized, i.e., they become the subclasses of C's parent class. The conflict resolution rules have to be applied and all the instances of these classes suitably modified. The instances of the deleted classes will also be deleted. The deletion of C also affects the composition hierarchy. For example, if the class Audio is deleted, then the deletion of all its instances also affects the instances of class Movie. Instances of class Movie will have invalid references to instances of Audio and thus need to be modified (typically, by invalidating the references). Thus, deletion of a class will also require a major reorganization of the information in the data base and hence the associated transaction is expected to be carried out with no other concurrently executing transactions. Note that deletion of a class is expected to be a relatively infrequent event and hence the associated transaction can be executed in isolation without severly affecting the system throughput.

5.7 Retrieval, Creation, Modification and Deletion of Classes – Multiple Inheritance

Although multiple inheritance is not considered a core object-oriented feature, several object-oriented systems support it. The presence of multiple inheritance complicates the issues of class creation, modification and deletion. We note that retrieval of information about classes is again similar to retrieval of instance objects. In systems that support multiple inheritance, a class may have more than one parent class. In this case, the class inherits the properties of all its parent classes. It inherits the instance variables and methods defined for each parent class.

The presence of multiple parent classes makes the concept of inheritance more complicated. For example consider the following class definitions:

```
CLASS House
   PARTS
     owner:Person
     size:Dimension
     external_color:Color
     internal_color:Color
     architect:Person
ENDCLASS.
```

```
CLASS Boat
   PARTS
     registry:String
     size:Dimension
     theEngine:Engine
     maxspeed:Integer
     manufacturer:Company
ENDCLASS.
```

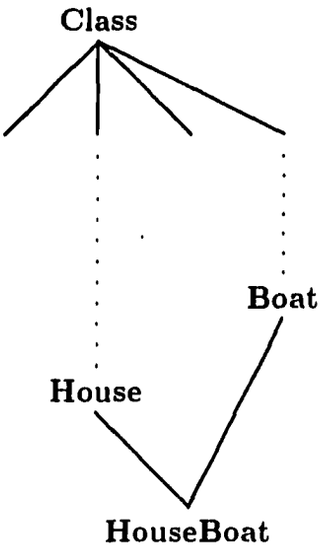

Figure 5.5: Multiple inheritance - partial class hierarchy

```
CLASS HouseBoat
   PARENTS
     House
     Boat
ENDCLASS.
```

The above definitions define a class HouseBoat that has the classes House and Boat as its parents (Figure 5.5). Thus, HouseBoat inherits the instance variables and methods defined for both its parent classes. However, this gives rise to a complication. Consider the instance variable size. Both the parent classes define the same instance variable. However, they both have a slightly different meaning. In class House, size refers to the size of the house. In class Boat, it refers to the size of the boat. Which of these definitions should hold for class HouseBoat? Similar complications arise when the parent classes have methods with the same name. To handle situations such as

the above, we propose the following conflict resolution rules for systems that support multiple inheritance.

Conflict Resolution Rules:

1. The parent classes of a class are ordered according to their position in the list of parent classes in the class definition.

2. If an instance variable or method is defined locally in a class, the local definition supercedes any inherited definition of the variable or method of the same name.

3. If an inherited instance variable or method from different parents has the same name, the one inherited from the parent listed first is retained.

Using these conflict resolution rules, the class HouseBoat in our example will inherit the instance variable size from the class House. The instance variable size defined in class Boat will not be available to class HouseBoat. Note that the use of multiple inheritance requires a thorough understanding of the class hierarchy. The conflict resolution rules sometimes create unexpected problems. For example, in class HouseBoat, the instance variable size is not inherited from class Boat. Therefore all methods inherited from class Boat and which refer to the instance variable size will have a potentially incorrect execution. Consequently, such methods will need to be changed appropriately. The effect of creation, modification and deletion of classes in the presence of multiple inheritance is described below.

5.7.1 Creation of classes

To create a new class, the data base user has to provide the definition for the new class. This definition includes the specification of the parent classes, locally defined

instance variables and methods.

When the new class is added as a leaf node in the class hierarchy, the impact on the class and class-composition hierarchies is minimal. When the new class is added, the conflict resolution rules for multiple inheritance are applied. The process of creating a new non-leaf class is more complex. Suppose that a new class C_k is inserted between the classes C_i and C_j where C_i is a parent of C_j. In this case, the entire class hierarchy rooted at C_j is affected. The definition of class C_j itself needs to be changed. This is because C_i is no longer (an immediate) parent of C_j. C_k replaces C_i as a parent in the definition of class C_j. The insertion of a new class not only affects the classes in the class hierarchy rooted at C_j, but also the instances of these classes. The insertion of C_k requires the application of conflict resolution rules to each class in the class hierarchy rooted at C_j. The application of the conflict resolution rules may invalidate some instance variables and at the same time some of the methods may need to be modified. The instances of the affected classes will also need modification. Similar to a system with single inheritance, typically the modification of these instances will require creating new instances with appropriate values (either copied over from the existing instances, or default values) and assignment of new object identifiers. The class-composition hierarchy is similarly affected. The instances of objects which refer to any of the instances of classes in the class hierarchy rooted at C_j may require modification. Furthermore, the methods of some of the classes which invoke methods from affected classes may also require modification. This is because some of the methods of classes in the class hierarchy rooted at C_j may no longer be available or may be modified.

5.7.2 Modification of classes

Modification of classes in the presence of multiple inheritance is another complex operation. Again, the impact on the data base system is minimal when the modified class is a leaf class, i.e., it has no subclasses. In this case, the instances of the modified class are also affected. When the class definition is changed, the conflict resolution rules for multiple inheritance have to be applied again. Furthermore, this modification may also affect other classes and their instances via the class-composition hierarchy.

If the modified class is not a leaf class, then the class hierarchy rooted at the modified class is affected. This affects the classes in the class hierarchy and the instances of these classes. The conflict resolution rules have to be applied successively to the modified class and all the classes in the class hierarchy rooted at the modified class. As a result of this, the instances of these classes may require to be modified along with some of the methods in the affected class. The modification of the class also affects the class-composition hierarchy. All the classes which refer to any of the classes in the class hierarchy rooted at the modified class may possibly need to be modified. Thus, the modification of a class may trigger changes over a large portion of the data base.

5.7.3 Deletion of classes

When the deleted class is a leaf class, the instances of the deleted class have to be deleted also. Furthermore, some other classes and their instances may be affected via the class-composition hierarchy. This may require the modification of methods in such classes which refer to instances and methods of the deleted class.

If the deleted class is a non-leaf class, the impact on the data base is greater. The

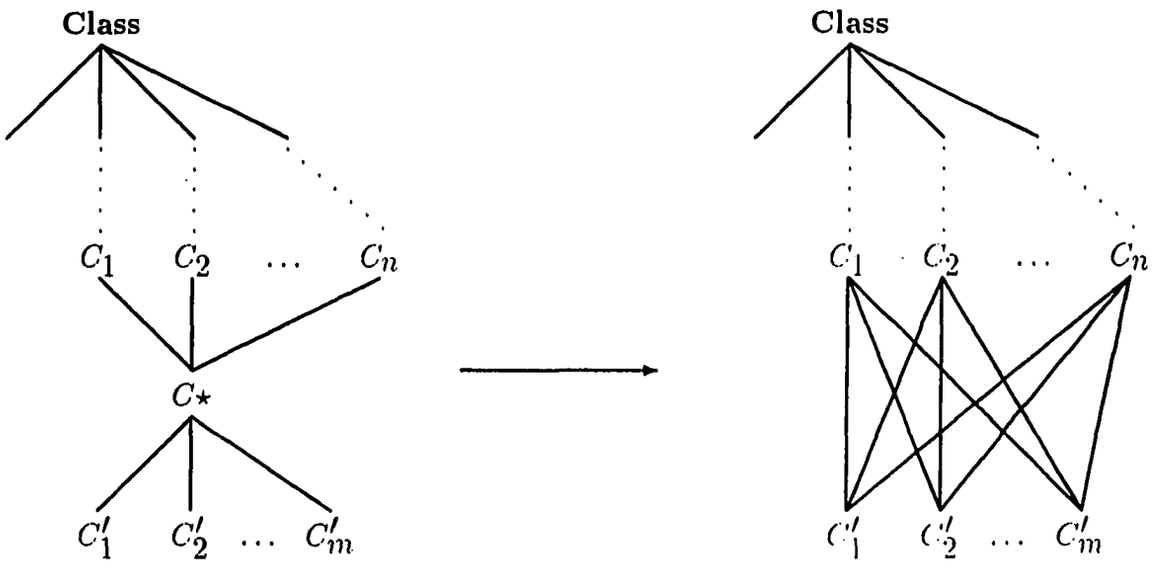

Figure 5.6: Multiple inheritance (deletion) - partial class hierarchy

instances of the deleted classes have to be deleted. Suppose the class $C\star$ is deleted. Let $C_{sub} = \{C'_1, C'_2, \ldots, C'_m\}$ be the subclasses of $C\star$ and $C_{par} = \{C_1, C_2, \ldots, C_n\}$ be the parents of $C\star$ (Figure 5.6). The deletion of $C\star$ requires the redefinition of all its subclasses. For each class $C_j \in C_{sub}$, each class in C_{par} now becomes a parent (instead of the deleted class $C\star$). The conflict resolution rules for multiple inheritance have to be applied successively to each class in the class hierarchy rooted at $C\star$. This may require the modification of some of the methods in these classes and also a modification of the instances of these classes. The class-composition hierarchy is also affected, requiring changes to the methods and instances which refer to the classes (and their instances) in the class hierarchy rooted at $C\star$. Deletion of a non-leaf class thus potentially affects a large portion of the data base.

5.8 Summary

In this chapter, we first described the research in the area of query languages for object-oriented data bases. The primary contribution of this chapter is the identification of the basic issues encountered by most query languages and possible solution to these problems. These issues include the translation of the query to operations on the underlying data base and the creation, deletion and modification of objects in the data base. The issue of modification of the structure of the data base, i.e., the class and the class-composition hierarchies, was also discussed.

Several other issues which are important in the context of queries on object-oriented data bases include indexing, clustering of objects and views. However, the treatment of these issues is beyond the scope of this research. In conclusion, queries provide an important means for users to access the data base in a meaningful and convenient manner. Object-oriented data bases, with their complex data relationships, provide a challenging opportunity for designers of query languages.

6. CONCLUSION

6.1 Summary

Object-oriented data bases have established themselves as a promising solution to the complex modeling requirements of advanced applications of this age. However, they may not be the best solution for every application. For example, some of the traditional data base applications, such as Banking, do not require the advanced features of object-oriented systems. In such cases, the overhead of an object-oriented system cannot be justified. On the other hand, applications such as CAD and MMDB systems need the advanced features of object-oriented systems. Thus, the suitability of object-oriented data bases is very much application dependent.

Transaction management is an integral aspect of any data base system. The nature of transactions in advanced applications, combined with the complications arising out of various object-oriented features, provide a new challenge to data base developers. Thus far, research in transaction management for object-oriented systems has primarily focussed on extending the concepts from relational systems. The need for a new transaction model is thus clear. This has provided the motivation for our research.

In this dissertation, we first identified the need for a new transaction model by describing the unique nature of transactions in advanced applications. The lack of

adequate research in this area was also noted. We then described a new transaction model for object-oriented data base systems. This model supports long-duration cooperating transactions and also addresses various object-oriented features. The model also exploits the semantic richness of the application domains and also the object-oriented systems. A new correctness criterion, known as correctability, was discussed. We also introduced the concepts of object-oriented correct and correctable schedules.

We also provided a concurrency control protocol for our transaction model. This protocol is optimistic in nature. It supports long-duration cooperating transactions and was shown to allow only correctable schedules. A distinctive feature of the protocol was the support for late binding, which is an important aspect of object-oriented systems. The users of a data base interact with it through queries. We identified some of the issues which are encountered by query languages and proposed a few solutions to these problems.

Our work in the area of transaction management for object-oriented data bases is an important contribution to the understanding of this research field. This work will aid the application developers to exploit the features of object-oriented systems. At the same time, the semantic content of the application itself can be used to enhance the effectiveness of the data base system.

6.2 Future Work

Research in the field of object-oriented data base systems will continue for a long time in the foreseeable future. As computer applications become more sophisticated, so does their need for complex modeling techniques. The richness of the object-

oriented model makes it a prime candidate for advanced application development. The following is a brief list of areas wherein significant amount of future work is expected to be continued.

- Defining standards

  Although a standard object-oriented data model is an unrealistic goal, attempts will be made to standardize some of the basic core concepts. We do not expect this to be a question of choice, but rather necessity. Adopting these standards will be the only means for diverse data bases to communicate with each other. The trend of increased office automation will make multi-data base communication a necessity.

- Exploiting semantic information

  General purpose data bases will continue to flourish in the future. However, there will be a distinct and increasing need for data bases that are customized for advanced applications. Such data bases will be expected to intelligently reflect the semantics of the application and be highly optimized.

- Cooperative workgroup environments

  The advances in computer communications have made it possible for groups of people to cooperate on a single project, irrespective of physical distances. They share the same resources and generate the same final product. Efficient management and utilization of resources in such an environment is an important challenge. Furthermore, tools that provide rapid prototyping and development environments will be needed.

- Query (r)evolution

  Query languages have shown a major improvement over the years. In future, graphical query languages will be commonplace. With advent of multimedia computers, other forms of queries for object-oriented data base systems are expected to emerge.

- Multimedia integration and standards

  Multimedia systems are already appearing in the market. However, their integration into the overall system is only beginning. Advances in computer communications will also require adoption of standards for exchange of multimedia information between heterogeneous systems.

- Reducing overhead

  The features of object-oriented data base systems are accompanied by the increased overhead of the system. Substantial research is expected to be directed towards improving the efficiency of these data bases. Techniques such as indexing are already being used to speed up the retrieval of stored objects. Significant amount of research is still needed in making these data bases more effective.

6.3 Conclusion

Object-oriented data bases present a promising alternative for advanced application development. The coming years will see an increase in their popularity. In this dissertation, we have addressed several important issues encountered by object-oriented data bases. Our approach for modeling long-duration cooperative transactions, incorporating the various features of object-oriented systems, will aid the development of advanced applications that use these data base systems.

BIBLIOGRAPHY

[1] C. Date. *An Introduction to Database Systems, Volume I, 4th ed.* Addison-Wesley, Reading, Massachussets, 1985.

[2] E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella. Object-oriented query languages: The notion and the issues. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):223–237, June 1992.

[3] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, volume 21(2), pages 383–392, San Diego, California, June 1992.

[4] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, volume 21(2), pages 393–402, San Diego, California, June 1992.

[5] W. Kim. *Introduction to Object-Oriented Databases.* The MIT Press. Cambridge, Massachussets. 1990.

[6] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query processing in the objectstore database system. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, volume 21(2), pages 403–412, San Diego, California, June 1992.

[7] F. Bancilhon. Object-oriented database systems. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 152–162, Austin, Texas, March 1988.

[8] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Pub. Co., Reading, Massachussets, 1987.

[9] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notion of consistency and predicate locks in database systems. *Communications of the ACM*, 19(11):624–633, November 1976.

[10] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* The MIT Press, Cambridge, Massachussets, 1985.

[11] C. Papadimitriou. *The Theory of database concurrency control.* Computer Science Press, Rockville, Maryland, 1986.

[12] D. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[13] C. Beeri, P. Bernstein, N. Goodman, M. Lai, and D. Shasha. A concurrency control theory for nested transactions. In *Proc. 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 45–62. Montreal, Canada, August 1983.

[14] P. Bernstein, J. Rothnie, N. Goodman, and C. Papadimitriou. The concurrency control mechanism of sdd-1: A system for distributed databases (the fully redundant case. *IEEE Trans. on Software Engineering*, SE-4(3):154–168, May 1978.

[15] K. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *1st ACM SIGACT-SIGOPS Symp. on the Principles of Distributed Computing*, pages 157–164, Ottawa, August 1982.

[16] T. Hadzilacos and V. Hadzilacos. Transaction synchronisation in object bases. *Journal of Comp. and System Sciences*, 43(1):2–24, 1991.

[17] F. Bancilhon, W. Kim, and H. Korth. A model of cad transactions. In *Proc. 11th Intl. Conf. on Very Large Data Bases*, pages 25–33. Stockholm, Sweden, August 1985.

[18] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O_2 object-oriented database system. Technical report, Altair Technical Report, August 1989.

[19] J. Banerjee et al. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1):3–26. January 1987.

[20] J. Banerjee, W. Kim, and H. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, volume 16(3), pages 311–322. San Francisco, California, May 1987.

[21] J. Banerjee, W. Kim, and K. Kim. Queries in object-oriented databases. In *Proc. 4th Intl. Conf. on Data Engineering*, Los Angeles, California, February 1988.

[22] J. Garza and W. Kim. Transaction management in an object-oriented database system. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, volume 17(3), pages 37–45, Chicago, Illinois, June 1988.

[23] W. Kim, K. Kim, and A. Dale. Indexing techniques for object-oriented databases. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Applications, and Databases*. Addison-Wesley, 1989.

[24] W. Kim. A model of queries for object-oriented databases. In *Proc. 15th Intl. Conf. on Very Large Data Bases*, pages 423–432, Amsterdam, the Netherlands, August 1989.

[25] W. Kim. Object-oriented databases: Definitions and research directions. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):327–341, Sept 1990.

[26] D. Maier and J. Stein. Indexing in an object-oriented dbms. In *Proc. Intl. Workshop on Object-Oriented Database Systems, Languages, and Applications*, Pacific Grove, California, September 1986.

[27] A. Purdy, D. Maier, and B. Schuchardt. Integrating an object server with other worlds. *ACM Transactions on Office Information Systems*, 5(1):27–47, January 1987.

[28] L. Rowe and M. Stonebraker. The postgres data model. In *Proc. 13th Intl. Conf. on Very Large Data Bases*, pages 83–95, Brighton, England, September 1987.

[29] D. Woelk and W. Kim. Multimedia information management in an object-oriented database system. In *Proc. 13th Intl. Conf. on Very Large Data Bases*, pages 319–329, Brighton, England, September 1987.

[30] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[31] N. Lynch. Multilevel atomicity – a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[32] A. Farrag and M. Ozsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[33] P. Shah and J. Wong. Transaction management in an object-oriented data base system. *To appear in Journal of Systems and Software*, 1993.

[34] P. Bernstein, N. Goodman, and V. Hadzilacos. Recovery algorithms for database systems. In *Proc. IFIP 9th World Computer Congress*, pages 799–807, Amsterdam, September 1983. North-Holland.

[35] V. Hadzilacos. An operational model for database system reliability. In *Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 244–256, Atlanta, Georgia, March 1983.

[36] H. Garcia-Molina, J. Kent, and J. Chung. An experimental evaluation of crash recovery mechanisms. In *Proc. 4th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 113–122, Portland, Oregon, March 1985.

[37] S. Abiteboul and A. Bonner. Objects and Views. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, volume 20(2), pages 238–247, Denver, Colorado, June 1991.

[38] D. Beech. A foundation for evolution from relational to object databases. In *Lecture Notes in Computer Science - Proc. Intl. Conf. on Extending Database Technology*, volume 303, pages 251–270, Venice, Italy, March 1988.

[39] S. Dar, N. Gehani, and H. Jagadish. CQL++: An SQL for a C++ based Object-Oriented DBMS. In *Lecture Notes in Computer Science - Proc. Intl. Conf. on Extending Database Technology*, volume 580, pages 201–216, Vienna, Austria, March 1992.

[40] D. Straube and M. Ozsu. Queries and query processing in object-oriented database systems. *ACM Transactions on Information Systems*, 8(4):387–430, October 1990.

[41] C. Lecluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, volume 17(3), pages 424–433, Chicago, Illinois, June 1988.